Notes on the Programming Language LISP

by Bernard Greenberg

Acknowledgements:

I would like to thank the Student Information Processing Board of MIT for the original idea of the course from which these notes developed, and their continuing support and enthusiasm for this project. I specifically would like to thank Lee Parks of SIPB for a tremendous amount of time and effort on preparing and editing this manuscript and helping with the writing of some parts. I would like to thank Dave Moon and others at the MIT Laboratory for Computer Science for reading the manuscript and offering many valuable suggestions, and Honeywell Information Systems for the time to carry this project out.

For the 1978 edition, I would again like to thank SIPB and Honeywell for support and time. I wish to thank Dan Weinreb of the MIT Artificial Intelligence Lab and Allan Wechsler of the MIT Joint Computer Facility for participating in this massive reorganization and rewrite.

Notes on the Programming Language LISP

PART 1

Introduction

LISP stands for LISt Processing language. It is a computer programming language possessing many capabilities lacking in most conventional languages. These special capabilities deal with manipulation of highly structured and symbolic information. Lisp has been used to great advantage in such fields as artificial intelligence research, symbolic mathematics systems such as MACSYMA, modeling and simulation, and computer language translators.

Lisp was invented in 1958 by John McCarthy, then of M.I.T. Originally implemented as a collection of FORTRAN subroutines on the IBM 7090, Lisp gained popularity, and was soon implemented on a variety of widely different machines.

In this presentation we will deal with the M.I.T. Artificial Intelligence Laboratory's Maclisp dialect of Lisp, which is available both on the DEC PDP-10 and on Honeywell's Multics.

Notes on the Programming Language LISP

Objects

All programming languages deal with manipulating information; however, various languages are better at manipulating certain kinds of information. Most languages, such as BASIC, FORTRAN, PL/I, ALGOL, and APL are best at dealing with such things as numbers, character strings, arrays, and pointers. These are computer-like beings, which live in registers and variables in computer programs. They are loaded, copied, stored, incremented, decremented, and concatenated.

LISP deals with a kind of being called an object. Objects have identity, i.e., being or uniqueness. The identity of an object is its most fundamental attribute, and cannot be changed. Two objects may resemble each other in all other attributes, and yet be different by virtue of their different identities. They are two different objects, as different as two identical twins, or two 1963 copper pennies. This is unlike the numbers and character strings of other languages: any 3.86 is the same as any other 3.86 in PL/I.

Lisp objects are often used to model real-world objects. Like real-world objects, Lisp objects have properties and relations to each other. A typical real-world object, like a house, has a color, a number of stories, the street it is on, the people who live in it, and other qualities and quantities as "properties". The street, in turn has many houses on it, a set of streets it crosses, a name, and forth. In a Lisp program, we might have one object represent each house we were dealing with, and another represent each street. Lisp allows us to define, establish, utilize and change the various properties and relations of groups of objects. It is in this way that Lisp programs can be written to model the behavior of real-world systems.

There are several types of Lisp objects. Objects of different types have different kinds of attributes and uses. All objects of all types live together in harmony in the list structure world. The objects describe each other and relate to each other in all kinds of interesting and dynamic ways, as desired by the programs at hand.

We will concern ourselves at first with the four most important types of objects: symbols, conses, fixnums, and subrs. Symbols and conses are used to represent the structuring of information; they are the "nouns" of Lisp. Fixnums are basically the integers of conventional languages. The only property a fixnum has its its "numeric value", or magnitude. Subrs are active beings who perform operations with objects. They are like procedures, functions, and operators in other languages. They are the "verbs" of Lisp.

Symbols

A symbol is an object which has three characteristics:

1) A printname, which is an arbitrary character string. For example: foo, a0126, first-item-in-list, !BBBBjhfkg. Normally, there is only one symbol in a given Lisp world with a given printname, because, as we will see, this greatly simplifies the task of writing programs. There are advanced means, however, to create different symbols with the same printname. This is not generally useful, but it does point out that the printname is a property of the symbol. "Fred" is not a symbol; it is a character string. "Fred" may, however, be the printname of a symbol.

2) A binding, which is some object in the list structure world. It can be any object, even the symbol itself. The word "binding" is a good one: it suggests that the object is on the end of a leash being held by the symbol. Any Lisp object can be the binding of one or many symbols, or perhaps of none at all. A symbol may either be bound to some object, or it may not have any binding at all. When a symbol does not have a binding, it is said to be unbound.

3) A property list, another object in the list structure world that is used to associate objects with this symbol in any way the programmer desires. For instance, the programmer can specify some other object as the "color" of a given object, another as its "affiliation" and a third as its "brother". We will learn more about property lists and how to use them in Chapter 2.

Often we will want to represent Lisp objects and the relations between them by pictures. We will draw symbols as stylized "atoms", because in the old days, symbols used to be called atoms. Now, the term atom is used to describe any object except a cons, which is the next type of object we learn about.

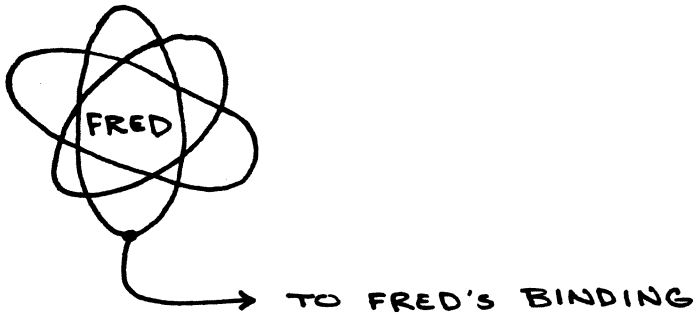
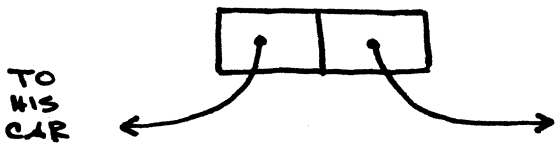


Figure 1. Picture of symbol.

CONSES

A cons



is an object which cares about two (not necessarily different, or even other!) objects in the list structure world. They are called its car and cdr (pronounced could'er). These terms originate from IBM 7090 addressing formats. Another common term for conses is cons-cells.

Here is a picture of a typical cons:

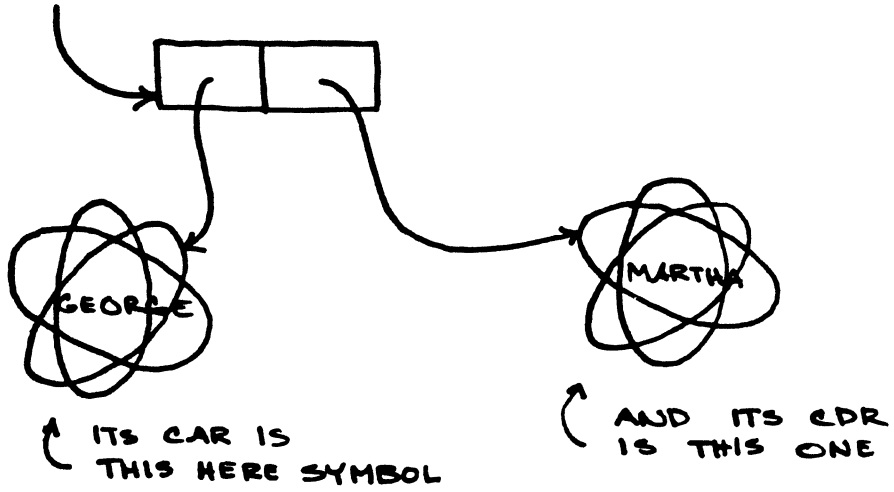


Figure 2. Picture of a cons.

FIXNUMS

A fixnum is a Lisp object whose interesting quality is its magnitude, an integer. There are ways to perform arithmetic on the magnitudes of fixnums. The results of these operations are usually fixnums, as well. Here is a picture of a typical fixnum:

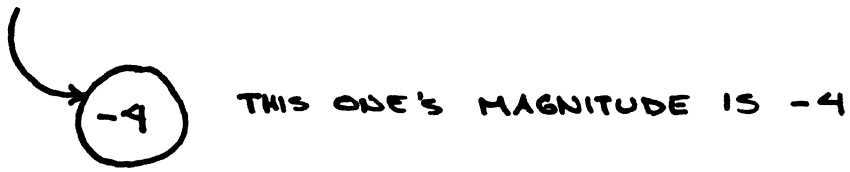


Figure 3. Picture of a fixnum.

EXAMPLES from the List Structure World

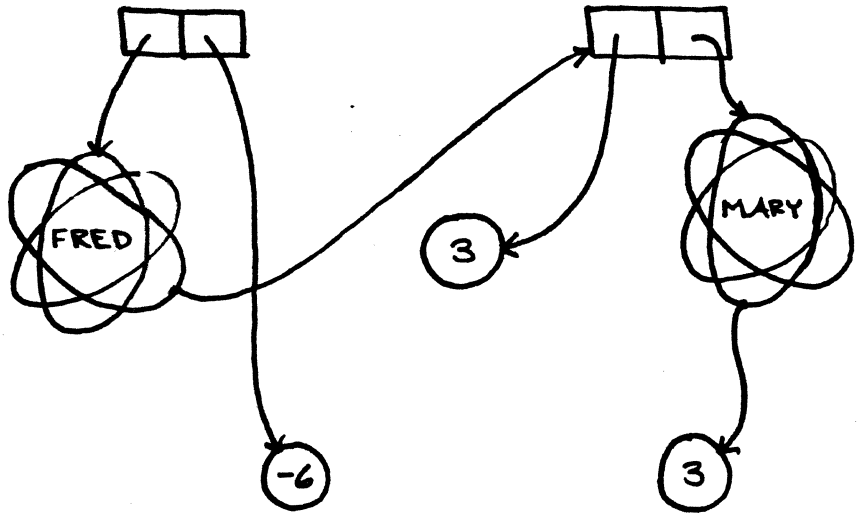


Figure 4.

The car of the cons in the upper left hand corner is the symbol with print-name "fred". The cdr of that cons is the fixnum at the bottom, whose magnitude is -6. The binding of the symbol named "fred" is the cons in the upper center. The car of that cons is the fixnum in the center, whose magnitude is 3; the cdr of that cons is the symbol named "mary". That symbol's binding is the fixnum on the right, whose magnitude is also 3.

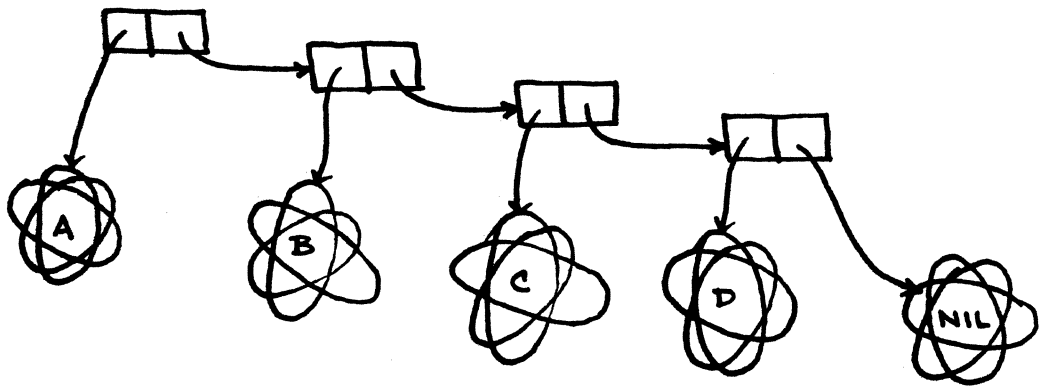


Figure 5.

The car of the first cons is the symbol whose print-name is "a"; the cdr of each cons is the next cons to the right, except the rightmost one, whose cdr is the symbol named "nil". A bunch of conses strung together by their cdrs like this is called a list; we will talk more about lists later.

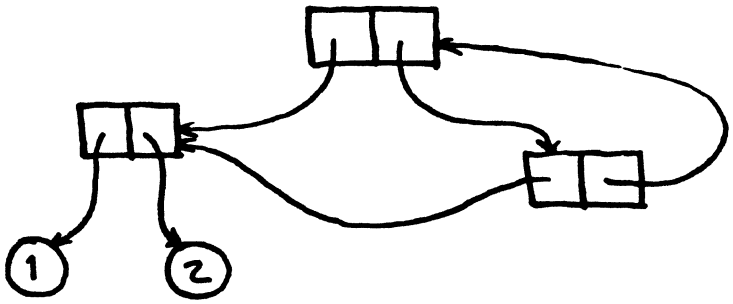


Figure 6.

The car of the cons at the top is the cons on the left; the cdr of the cons on the top is the cons on the right. The car of the cons on the left is a fixnum whose magnitude is 1, and its cdr is a fixnum whose magnitude is 2. And so forth.

Subrs

Lisp programs are built by writing functions that deal with objects. Like procedures in other languages, functions call or invoke each other. They pass objects around; objects to indicate what to do, objects to do it to, objects to say what was done. One function calls the second, passing it objects as input, as arguments. This is called applying the second function to those arguments. When the second function is finished doing what it was supposed to, it passes one object back to the first function. This is called returning this object as a value. The returned value may be the "answer" of the second function, or perhaps some indication of how well it performed the task it was asked to do.

The ability to have functions call each other, and pass the result of one on to the next, i.e., functional composition, is one of the most characteristic features of Lisp. Functions are built up by specifying calls to other functions, and so forth. Now, if any of these functions are to do anything but call each other, we will need some "built-in" operations that perform their services without our having to tell them how. Indeed, Lisp has such functions. They are called subrs (pronounced "subber"s). They are like the "operators" of FORTRAN or BASIC (e.g., "+", "-", etc.) or the "builtin functions" of PL/I (e.g., "substr", "index"). Like all functions, they are called by applying them to arguments, which are objects, and they return an

object as a value to the function which called them when they are done.

Subrs create, examine, and modify objects in the List structure world. Subrs can also communicate with the outside world, through input/output devices such as the terminal.

Lisp books and manuals have names for subrs, so that we may talk about them and learn their properties. The names don't exist in the list structure world, but later we'll see how to get at subrs by their names. Bear in mind that subrs are Lisp objects, and have identity.

Some Fundamental Subrs.

1) car — This subr may be applied only to a cons. When you apply the car subr to a cons, the result you get back is that cons's car.

2) cdr — This is just like car, but it returns the cdr of the cons to which it is applied.

3) cons — This subr causes a brand-new cons to be created. The car and cdr of that cons will be, respectively, the first and second arguments presented to the cons subr. The value returned by the cons subr is the cons it created.

We have just learned about three elementary subrs for creating and examining conses. Next we present two subrs for modifying already existent conses, and for examining and modifying symbols. These operations are not very common in elementary Lisp; creating symbols is even less common. We show you these subrs here for the sake of completeness, because conses and symbols can be modified, and symbols inspected.

4) rplaca (for RePLACe cAr, prononounced "replocka"). rplaca is applied to two objects. The first must be a cons, and the second may be any object. The first argument, the one which is always a cons, is altered such that the second object is now its car. This, of course, in no way affects whatever object that used to be its car. It's like changing one's shirt or shoes. The first argument (the cons), now changed, is the value which is returned.

5) rplacd (pronounced "replockda"). Like rplaca, rplacd takes a cons and anything as arguments but makes the second object be the cdr of the first.

6) symeval This subr may only be applied to a symbol. The object returned is the binding of that symbol.

7) set Takes two arguments, the first of which must be a symbol. The second argument is made to be the binding of the first argument. The value that the set subr returns is that object which was its second argument.

There are subrs for dealing with fixnums, performing computations with their magnitudes, and producing fixnums, as returned values, with magnitudes indicative of the result of arithmetic operations to be performed. For example, there is a subr called "+" which does addition: it takes some fixnums, figures out the sum of their magnitudes, and returns as its result a new fixnum, whose magnitude is that sum. They are simple, and used very often.

Here are some of these "numeric" subrs.

8) + This takes any number of fixnums, and returns the sum of its arguments. Here we are speaking loosely, and really mean that it returns a fixnum whose magnitude is the sum of the magnitudes of its arguments. When speaking of numeric subrs, we shall continue to speak this way.

9) - This returns the difference of its arguments.

10) * This returns the product of its arguments.

11) / This returns the quotient of its arguments.

There is a certain symbol whose print-name is "nil" which is treated specially by many subrs. It is of primal importance in Lisp because of this. When we mention "nil" in this text, it is this symbol that we will mean.

Lists

One of the most common kinds of data structure created in Lisp programs is the list. As we know, a cons can specify two objects, namely those which are its car and its cdr. We need the ability to specify a set of an arbitrary number of objects. The way this is done is to build a chain of conses, strung together by their cdrs. The cdr of the last cons is "nil".

Go back to illustration 5; it portrayed a list of four elements: the symbols named "a", "b", "c", and "d". It is an ordered set; "a" is the first element, "b" is the second, etc. The car of the nth cons in the chain is the nth element of the list. The "nil" at the end is not an element of the list; it is there because something has to be there; if a cons were there, it would not be the end of the

list. The presence of "nil" is a convention to signify the end of a list.

A list is but one example of a data structure built out of lower-level data structures. When such structures are built, the attributes of the lower level data structures, i.e., the conses, take on new meaning in terms of the higher level data structure, i.e., the list. The first cons of a list can be thought of as the list itself; thus, the car of that cons is the first element of the list, and the cdr of that cons is the "rest" of the list. As a matter of fact, we usually speak of "the car of a list" when we mean "the car of the first cons of a list", i.e., its first element, and "the cdr of a list" when we mean "the rest of a list".

A single cons whose cdr is "nil" can be thought of as a list of one element, that element being the object which is the cons's car. Similarly, the symbol "nil" itself can be thought of as a list of zero elements, for the cdr of a list of n elements is a list of n-1 elements. "But suppose you want to have a list with nothing but the symbol 'nil' in it" the overzealous student poses. Well, such a list looks like this:

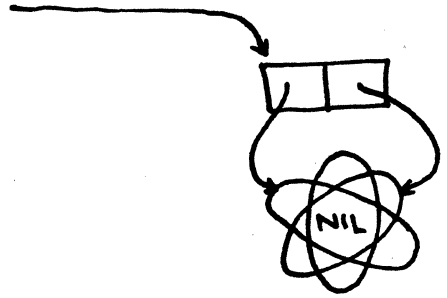


Figure 7.

just as the rightmost cons in figure 5 is a list of one element, the symbol "d". There is a big difference between an object, and a list of that object. The latter is a cons whose car is the former, and whose cdr is "nil".

Lists aren't really the same as mathematical sets, because a set cannot contain an object more than once, and the elements of a set are not in any particular order. Here is a list of the symbol

"albert", the symbol "max", and the symbol "albert" again.

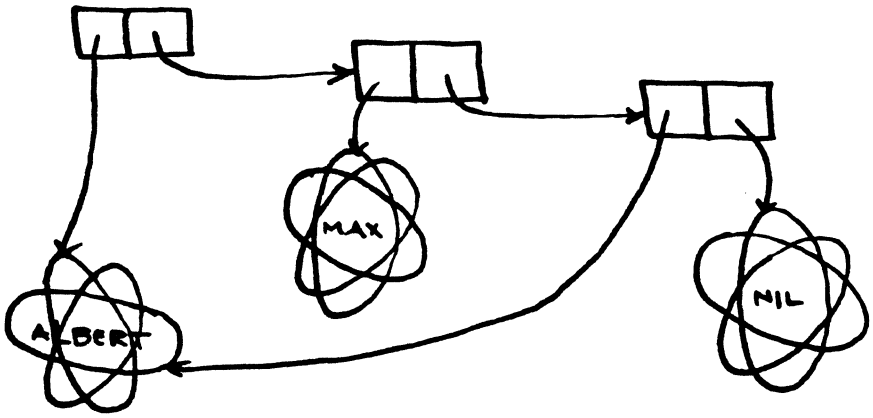


Figure 8.

- - - NOTICE - - -

We have been talking about Lisp objects and their structure for several pages now. It is instructive to stop at this point and compare the basics of Lisp with the basics of most other programming languages. For example, were we describing ALGOL instead of Lisp, we would already be talking about programs, having glossed over whatever issues of data-structure could be raised. The mechanics of creating programs in Lisp are indeed a major part of the language; yet, the structure and semantics of those programs cannot be understood without comprehension of the very real and specialized world of the data objects upon which they operate.

In order to deal conveniently with large classes of structures in the list-structure world, we will have to do better than drawing little pictures of boxes. There exists a representation of objects in the form of printed text; this is called the printed representation. For example, the printed representation of

Figure 9.

is

foo

That is, the printed representation of a symbol is its

print-name. Here is another example: the printed representation of

Figure 10.

is

54

That is, the printed representation of a fixnum is the numeral representing its magnitude. (1)

The printed representation of a cons is more complicated. In the simplest case, it is an OPEN PARENTHESIS ("("), the printed representation of its car, a DOT ("."), the printed representation of its cdr, and, finally, a CLOSE PARENTHESIS (")"). For example, the printed representation of

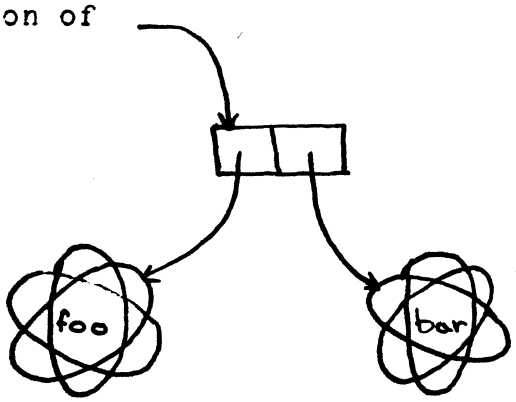


Figure 11.

is

(foo . bar)

(1) We might as well point out at this time that Maclisp deals with numbers, unless you explicitly request otherwise, in octal (base 8). Although there are ways of changing this, the eager novice must be wary of this when he complains that Lisp thinks the sum of 4 and 5 is 11.

Or, for example,

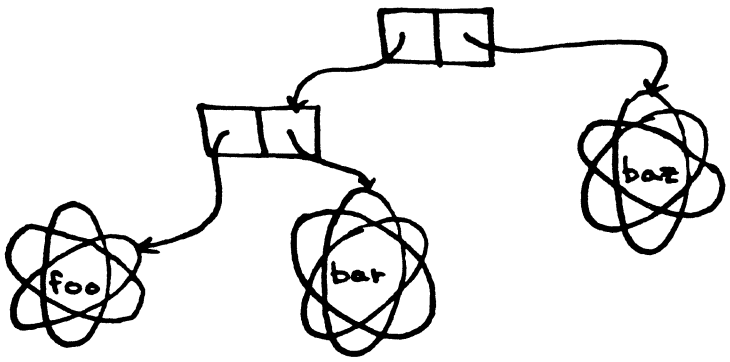


Figure 12.

has the printed representation

((foo . bar) . baz)

Note that the printed line above, which appears to be the printed representation of the data structure drawn above it, may as well be viewed as the printed representation of the cons at the head of that data structure. The cons is an element of the list-structure world; the data structure is an element of the conceptual data world of the problem we are programming. The identification of that cons with the data structure is an important one; the concept of building data structures out of conses and atomic objects (symbols and fixnums) is the basic data-building technique of Lisp.

This rapidly gets tedious for conses like the one in figure 5; this useful list of symbols would have the awkward representation

(a . (b . (c . (d . nil))))

The people who started Lisp were really into lists; hence the name of the language. So the printed representation became optimized to the representation of lists. Thus, the printed representation of the cons at the head of the list in Figure 5 is

(a b c d)

Recall that a list is a chain of conses hooked by their cdrs, with the cdr of the last cons being nil. The printed representation of a cons whose cdr is nil doesn't have the DOT or the "nil". Thus we never write "(a . nil)"; instead, we write "(a)".

It would appear that there are two distinct printed representations for conses, depending upon whether or not the cons is the head of a list. In fact, there is only one. Here is how you get

it; the printed representation of any cons starts with an OPEN PARENTHESIS, followed by the printed representation of its car, e.g., "(a ". What comes next depends on the cdr. If the cdr is nil, we continue with a CLOSE PARENTHESIS, and that is all. If the cdr is a cons, we continue with the printed representation of that cons, without its leading open parenthesis. If the cdr is anything else, we continue with a DOT, the printed representation of that cdr, and finally a CLOSE PARENTHESIS.

It isn't really necessary to understand this algorithm completely at this point; you will soon get a feel for the correspondence between list structure and its printed representation. To make it clearer, here are a bunch of illustrative examples:

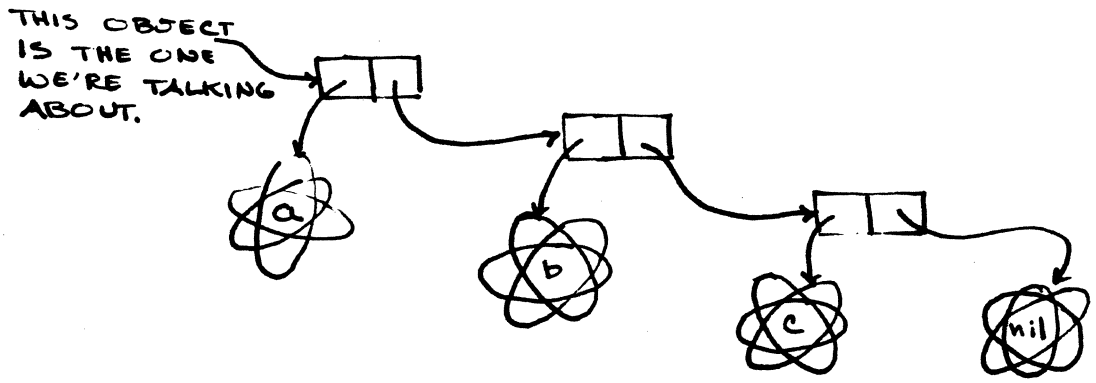


Figure 13. (a b c)

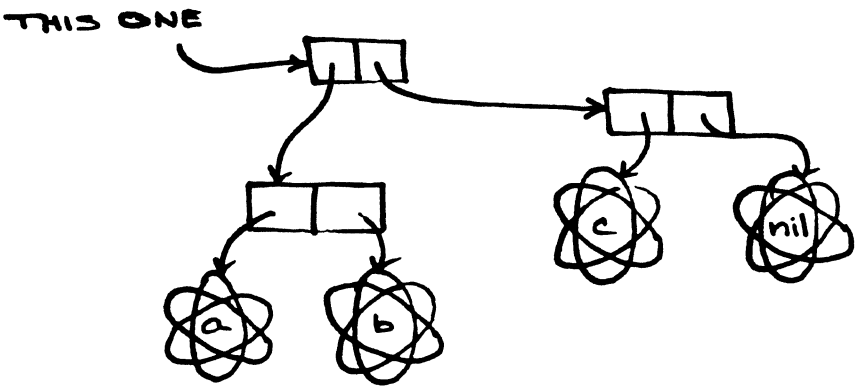


Figure 14. ((a . b) c)

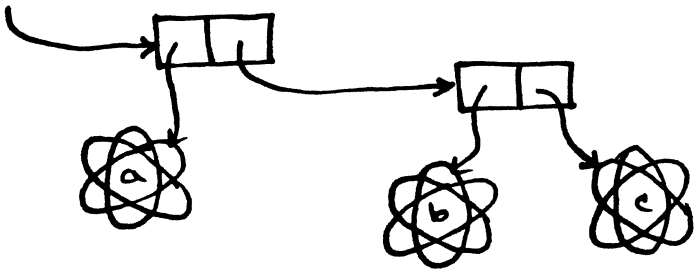


Figure 15. (a b . c)

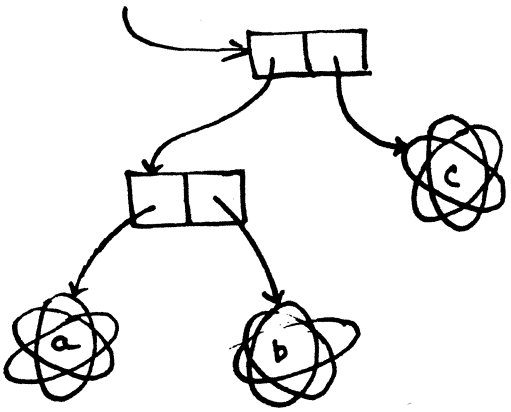


Figure 16. ((a . b) . c)

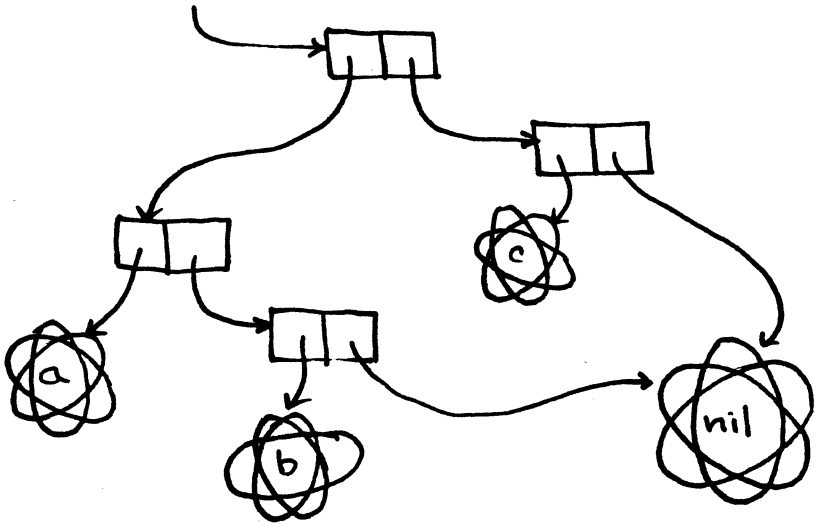


Figure 17. ((a b) c)

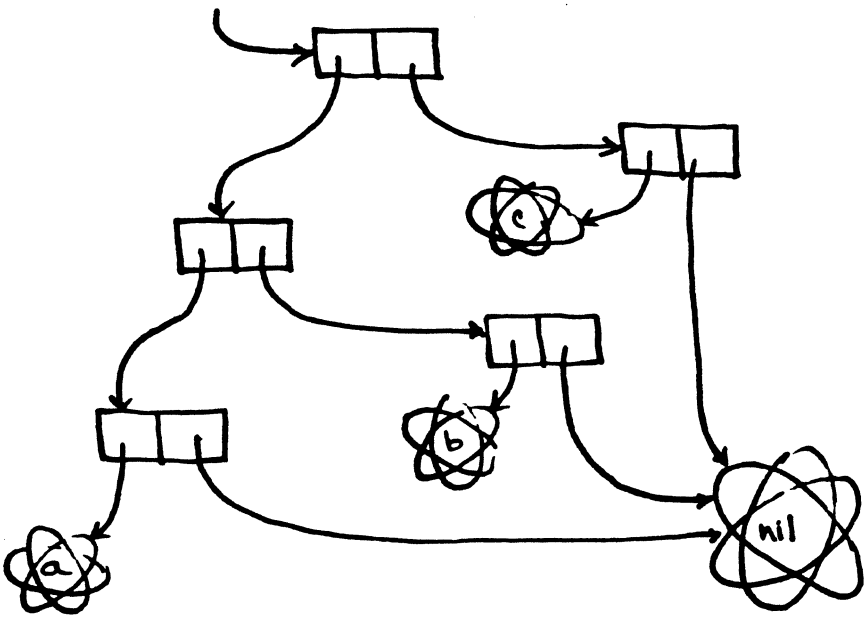


Figure 18. (((a) b) c)

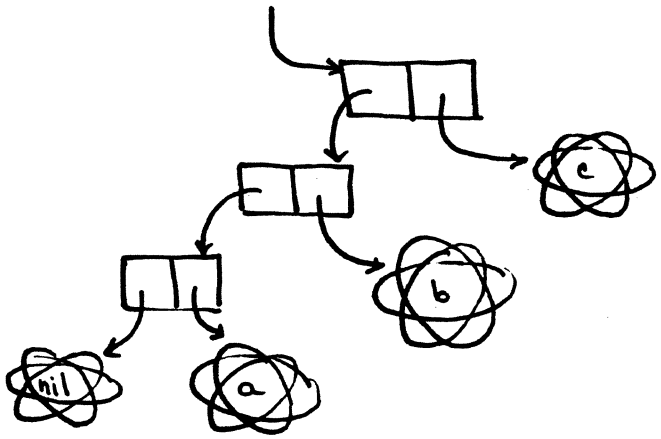
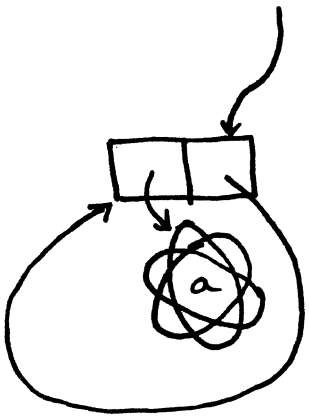


Figure 19. (((nil . a) . b) . c)

Note that some objects have infinite printed representation. For example, the printed representation of



is

(a a a a a a a a a a a a a a ...

and so on forever.

By the way, subrs also have printed representations, but you

don't see them very often and they are kind of ugly. On Multics they look like

#000355402443000402000000

and on ITS they look like

#13011

As we mentioned, there are subrs which communicate with the outside world about objects in the list-structure world:

12) print Subr of one argument. Types out, on your console, the printed representation of its single argument. It returns as a result a symbol whose print-name is "t". (Don't worry about the "t" right now; we'll get back to it later.)

The READER

Printed representations suggest a good way to create list structure; we can envision reading the characters of a printed representation from the terminal, and creating corresponding objects. The subr that builds a Lisp object from its printed representation is read. Read is basically the inverse of print.

13) read A subr of no arguments. It reads in the printed representation of a single Lisp object, builds up such an object, and returns the object.

When read sees the printed representation of a cons, it creates a brand-new, never-before-seen cons. This is one of the ways new conses get into the list-structure world.

When read sees the printed representation of a symbol (that is, when it sees something that looks like the print-name of a symbol) it basically creates a new symbol with that print-name, with no binding. However, read first looks to see if it has ever created a symbol with that print-name before, and if so, uses the already existing symbol instead of making a new one. It looks up the print-name on a big table of such symbols called the obarray. The obarray is a catalog, or registry, of all symbols that read has ever created. This means that every time the character string "foo" is read in, we get the same symbol. Symbols that are registered on the obarray are said to be interned. All symbols returned by the reader are interned symbols.

When read sees something that looks like the printed representation of a fixnum, that is, something that looks like an integer, it creates a fixnum of the appropriate magnitude. This is how numeric data is entered into Lisp programs.

The reader will not create subrs; if it sees the printed representation of a subr, it will create a symbol with that print-name. We do not want the reader to create or identify subrs by their printed representation (that strange "#13011" thing we saw above), because it is not very meaningful to the Lisp programmer. Shortly we will learn how to specify subrs by better means.

Remember before when we pointed out some list structure whose printed representation is infinite? read will never produce this kind of stuff, for the simple reason that this printed representation cannot be typed in! It is possible for two different objects of markedly different structure to have identical printed representations: read will only produce one of them (obviously, when you type that printed representation in, there is only one thing read will give you). Read never produces list structure with any particular cons being the car or cdr of more than one other cons. Thus, when read sees "((a . b) (a . b))", it produces

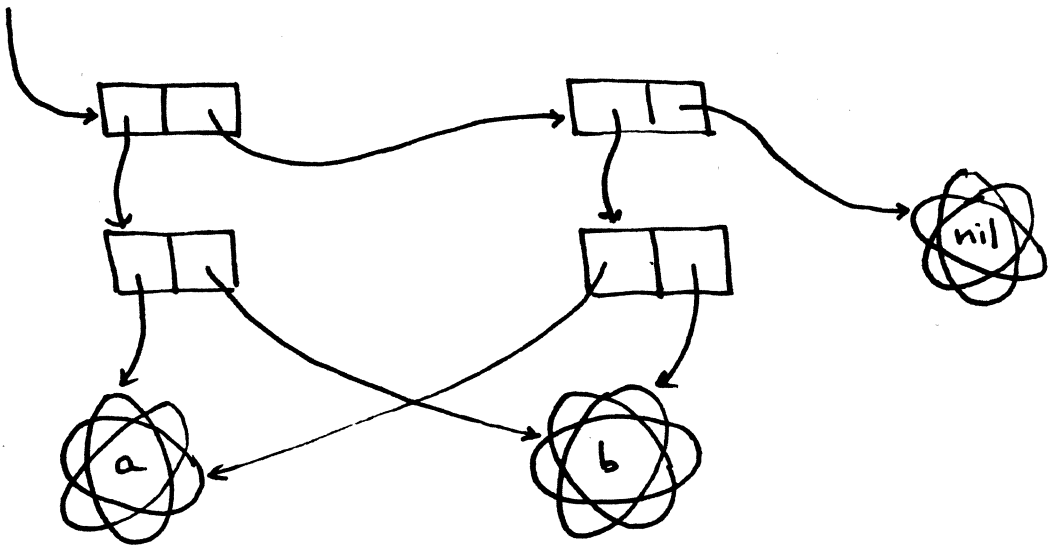


Figure 20.

and not

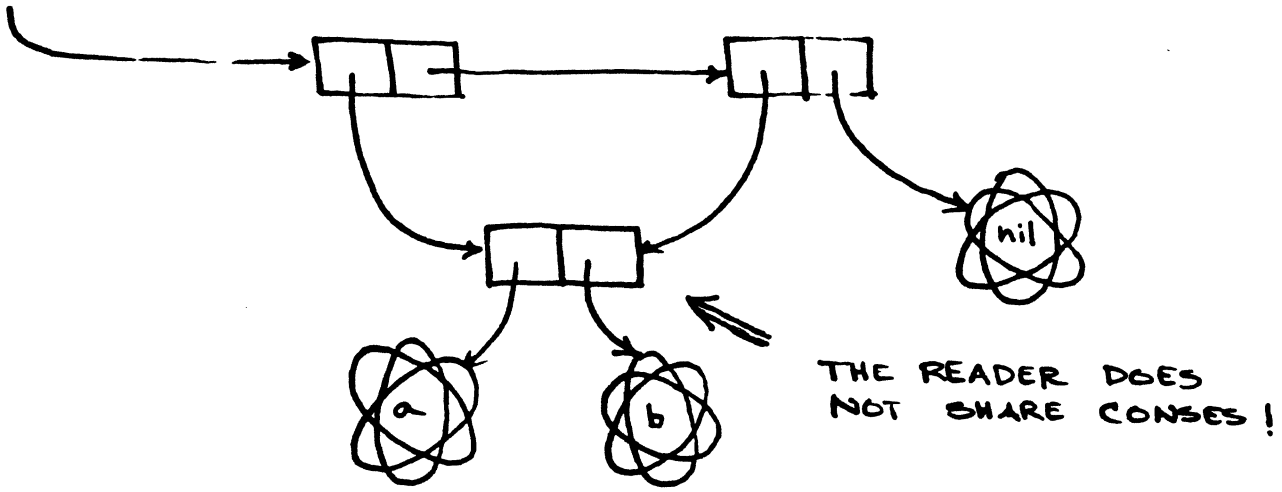


Figure 21.

nor

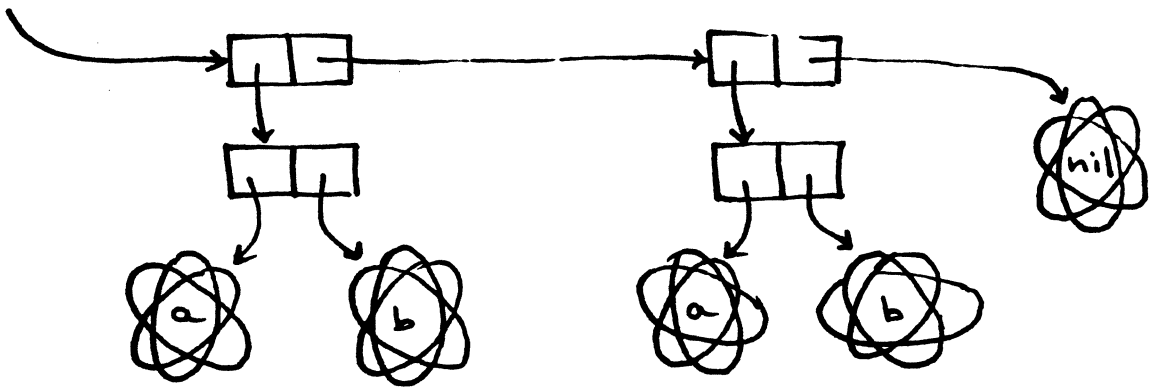


Figure 22.

Programs

Here are examples from some programming languages:

```
eppap     sst|cme.devadd,.cme
stca      ap|0,74
tsx0      pc_trace$read
tra       read_exit
```

```
sp -> symbol.token = htp -> h(i).fp;
if sp -> symbol.attributes.structure then do;
```

```
//SET1 DD DSN=MYDATA,DCB=(RECFM=FB,LRECL=80),SPACE=(,,1)
```

```
ei/baz/zj-:s/foo/3deo/bletch/eq$$
```

In Lisp, we would like to represent expressions and programs as list structure. For example, the FORTRAN expression 3+4 can be represented by the list structure written as

```
(+ 3 4)
```

and a*b+c*d as

```
(+ (* a b) (* c d))
```

A very nice thing about this is that the list structure corresponds exactly to the logical structure of the expression. No "precedence rules" are needed at all.

As you can see, it is a convention to write the operator first, followed by all of its operands. This is more general than the usual "infix" notation used in most languages ("infix" is where you put the operator in between the arguments, as in "foo + bar * baz"), because it facilitates operators with any arbitrary number of arguments, such as none, one, or five. Being able to say that the first element of a list that represents an expression represents the operator makes life simpler.

Notice how we are representing the addition operation with the symbol whose print-name is "+", and how we are representing mathematical variables by the symbols named a, b, c, and d. What we have ostensibly is a representation of a mathematical expression, by list-structure. In fact, this is one of the things Lisp was developed for, and it is not surprising that it is natural and convenient.

Lisp has the ability to compute the values of expressions. There exists a subr named eval which, given a list representing an

expression, such as the one above, will compute the "value" of that expression and return a Lisp object representing that value.

Eval is central to Lisp. It is eval which makes Lisp more than just a way of representing data. The magic wand of eval transforms Lisp into a programming language.

When handed to eval, an expression becomes a call to action, a specification of something to do. In this sense, "(+ (* a b) (* c d))" says: "Go out and multiply a by b, and then multiply c by d, and then add the two products". We have also expressed an order in which to perform these operations, by our arrangements of the operands. It is clear that we had to perform the two multiplications in order to obtain the addends, and so we had to multiply before adding.

Taking a piece of list structure and performing the operations specified therein is called evaluation. This is what eval does, and is the essence of the programming language Lisp. A program, in any language, is nothing more than a specification of a bunch of computations to be carried out in a certain order. In Lisp, programs are represented by list structure, and the process of executing them is evaluation.

Note that the representation of programs in Lisp is simply one use of the Lisp data world. ALGOL programs are represented in character strings, but ALGOL cannot even deal with character strings! Lisp programs can write, manipulate, and even debug other Lisp programs, or even themselves! It is this ability to manipulate its own programs as data, among other things, from which Lisp derives its substantial power.

In this way, list structure is used in Lisp to represent all computations: arithmetic operations with fixnums as well as structural operations such as finding the car of a cons or the binding of a symbol: in short, all of the operations perfomed by subrs. We use eval to cause these subrs to do their things in the list-structure world. A Lisp program is a collection of instructions to eval to apply subrs to Lisp objects. A Lisp program is also a Lisp object, which is interpreted by eval (itself a subr) to cause these instructions, these applications of subrs, to be carried out.

A piece of list structure that represents a computation, and is to be handed to eval to effect this computation, is called a form. The "(+ (* a b) (* c d))" which we saw earlier is a typical form. Lisp forms can represent any computation capable of being carried out, and thus Lisp can perform any such computation.

14) eval takes one argument, which is a form. It evaluates the form, and returns the result.

Evaluation is a well-defined procedure, and we must learn its details before we can write forms.

Exactly how does eval evaluate a given form?

Well, here is one simple case. Since we use fixnums to represent integers, the value of the computation represented by a fixnum with magnitude 3 is a fixnum with magnitude 3. So, when you give eval a fixnum, it just gives you back that fixnum. We say that fixnums self-evaluate.

To instruct eval to apply some subr to some objects, we first need a way to talk about the subr. The problem is, we cannot "type in the subr" itself; we want to be able to type in its name. For this reason, subrs are associated with symbols whose print-names are the names of the subrs. For example, the "*" subr is associated with a symbol named "*". The "*" subr may be found from the property list of this symbol; we haven't explained property lists yet, and this is not the place to do so. But there is a way of getting from that symbol to that subr, so when we want to apply the "*" subr to something, we can use the symbol named "*" to express this.

Let us call this symbol the "name-symbol" of the subr. Thus, the name-symbol for the "car" subr is a symbol whose print-name is "car", and which has attached to it the "car" subr. All name-symbols are interned, i.e., on the obarray, so that when the reader sees the character string "car", it will find the name-symbol for the "car" subr. That is how we "type in the subr"!

Here is how we tell eval to apply a subr to some objects: we give it a list whose first element is the name-symbol of the subr. The remaining elements are forms, which eval is to evaluate, as separate evaluations, to obtain the objects to give to the subr. The subr's result will be the result of the original evaluation. Read these three sentences several times, and then several times more: they are the heart of evaluation, which is itself the heart of Lisp.

For instance, if we wish to find the sum of 3 and 4, we can give eval a list, whose first element is the symbol "+" (which is the name-symbol of the "+" subr, which does addition), and whose second and third elements are fixnums of magnitude 3 and 4. Such a list has a printed representation

(+ 3 4)

If we give this list to eval, it will evaluate the fixnum "3" and obtain that fixnum again; then evaluate the fixnum "4" and obtain that fixnum, apply the "+" subr, found from the name-symbol "+", to those two objects, and obtain its result. This result is the result of the entire evaluation: a fixnum of magnitude 7.

Here is another example. Let us apply eval to a list which

prints as

```
(* (+ 3 4) (- 1 2))
```

This is an order to apply the "*" subr to two objects. Eval obtains the first of these objects by evaluating that sub-form which shows up as "(+ 3 4)". As we saw in the previous example, this produced a fixnum "7". Similarly, eval gets the second object by evaluating the "(- 1 2)" sub-form, obtaining a fixnum, "-1". Eval applies the "*" subr to these two fixnums, and the "*" subr returns a fixnum "-7", which eval returns.

We now see how we can build forms of arbitrary complexity, such as

```
(- (* 3 4 5) (+ (- 4 5) (* 5 6 7 2) (* 5 4 (- 4 3))))
```

So far we have only dealt with applying numeric subrs to fixnums. The same mechanism can be used to construct forms which apply any of the subrs we have learned about to any Lisp objects. For instance, we can ask for the car of a cons of the symbols "a" and "b", by applying the "car" subr to such a cons.

We must now construct a form which asks eval to apply the "car" subr to such a cons. Our first attempt at this might be something that looked like this:

```
(car (a . b))
```

This seems to be the right thing: the reader will indeed create symbols a and b if there are not already such on the obarray, and a cons with them as its car and cdr. The symbol "car" that the reader will find will have the "car" subr attached to it.

Close, but no cigar. Upon seeing this form, eval will correctly conclude that it is a request to apply the "car" subr to some object. That object will be obtained by evaluating the form "(a . b)". This is an ill-formed request to apply some function named "a". Not only is there no subr attached to the symbol "a", but even if there were, this is certainly not what we want. We are trying to apply "car" to the object "(a . b)", not the object which results from evaluating "(a . b)".

This confusion is only possible because Lisp forms, which are Lisp programs, are built of the same stuff, and print out the same way, as "data" objects in the list-structure world. We need a way of telling eval, "Don't evaluate this object, just return it. Your answer is not to be the result of evaluating this; your answer is to be this itself."

This is very much like trying to print out the string "X + 3" in BASIC or PL/I, by a statement like

```
120 PRINT X + 3
```

In BASIC, this will print out a number three greater than X. If a BASIC programmer wanted to print out the string "X + 3", he would say:

```
120 PRINT "X + 3"
```

The issue here is one of differentiating between a name for something and the thing itself. Lisp has such a mechanism for "quoting": it is called quote. Here is how it works: to ask for the car of "(a . b)", we give eval

```
(car (quote (a . b)))
```

Here is how this form is evaluated: eval sees that this is a request to apply the "car" subr to an object. This object will be found by evaluating

```
(quote (a . b))
```

Here is how this form is evaluated: eval sees that this is a request to apply the quote subr to an object. BUT, eval knows that the quote subr is one of a very special class of things called fsubrs. A "Funny SUBR" is really a piece of the evaluator- it is something which works on forms as part of the business of interpreting Lisp as opposed to operating on the objects in the Lisp world that represent the programmer's data. An fsubr is not really a subr at all. Seeing the request to "apply" quote, eval does special things with the form in which quote appears, instead of evaluating parts of it to get the objects to which quote is to be applied. These special actions are those associated with "quote": for this reasons, forms of fsubrs are often called special forms: eval's actions in evaluating each kind of special form differ.

In the case of quote, eval takes the second element of the form in which quote appears, and returns it as the result of the evaluation. In this case, the special form containing quote is

```
(quote (a . b))
```

and that second element is

```
(a . b)
```

This is the result of the evaluation; it is the object to which eval now will apply the "car" subr.

Thus, we do not ask "what does quote do"? quote is an internal part of the evaluator; what it does is an internal feature of the implementation. What we want to ask is "what does the evaluator do

with a "quote" form? This is the appropriate question for all special forms. In the case of "quote", the answer is, "The second element of the form is returned as the result of the evaluation".

In this way, we use quote to incorporate "constants", or pieces of the program itself, into the running world of our Lisp program.

Thus the result of evaluating "(quote (a . b)) is (a . b). This is the object to which eval now applies the "car" subr. The "car" subr, given this object, now returns the object's car, namely,

a

This is the desired result of evaluating "(car (quote (a . b)))", i.e., obtaining the car of "(a . b)". Notice that we could have applied the "+" subr to "3" and "4" by evaluating

(+ (quote 3) (quote 4))

The only reason we didn't have to say this is that fixnums evaluate to themselves. List structure is not self-evaluating. Thus, "quote" provides a mechanism for putting into Lisp programs pieces of list structure for your program to work with, instead of for eval to work with. (Your program itself is just a piece of list structure, which eval works with). Such pieces of list-structure "data" in programs are sometimes called constants. "quote" provides the mechanism for specifying list-structure constants in programs.

15) quote causes eval to return the second element in the form in which quote is used.

It would soon get boring if all we could do is perform computations on constants, as seen from the viewpoint of a computer programmer. We could have used a pocket calculator to do what we have done so far. Finding the cars of lists we made up for the fun of finding their cars does not have much promise either. We need a way to work with objects which are not known at the time we write the program: we need something like the variables of other programming languages. Lisp has such a notion of variables.

Symbols may be used as variables. The value of a symbol, when used as a variable, is that object which is its binding. Thus, when you give eval a symbol, you get back its binding. A symbol is said to evaluate to its binding. It is an error to attempt to evaluate an unbound symbol. Thus, we can set the values of variables by giving symbols bindings, and use the values of variables by using these symbols in forms.

Suppose we try to evaluate "(+ a b 3)", (1) where the binding of "a" is a fixnum 4 and the binding of "b" is a fixnum -2. Eval evaluates the "a", and gets a "4", it evaluates "b" and gets a "-2", and evaluates the "3" and gets a "3". Then it applies the "+" subr to the three fixnums which it got, and "+" gives it back a "5". Eval returns the "5".

In order to assign values to variables, all we need do is assign bindings to symbols. We have already described a means for doing this; it is the "set" subr. If we wish to bind the symbol "a" to the symbol "b", we apply "set" to "a" and "b". Go back to the description of "set" earlier if this is at all unclear. So, we can ask eval to perform such an application for us by giving it

```
(set (quote a) (quote b))
```

The function of the "quote"s ought to be clear. We want to give "set" the symbols "a" and "b", not the result of evaluating them. In contrast, if we wanted to assign the value of "b" to "a", that is, make the binding of "a" be the same as the binding of "b", we would give eval

```
(set (quote a) b)
```

After the evaluation, "a" and "b" will be bound to the same thing. Some more examples: we can make the binding of "a" be a fixnum "3" and the binding of "b" be a fixnum "4" by giving eval the forms

```
(set (quote a) 3)
(set (quote b) 4)
```

Having done this, evaluation of the form

```
(+ a b)
```

would give "+" the fixnums "3" and "4", and it would yield a fixnum "7". By contrast, evaluation of the form

```
(+ (quote a) (quote b))
```

would quickly cause an error, since "+" will be handed the symbols "a" and "b", and "+" deals only with fixnums, not with symbols.

In practice, "set" is almost never used. Just about every time we would want to use <u>set</u>, we want to say

---

(1) By "(+ a b 3)" we really mean "a list whose printed representation is (+ a b 3)". This is simply a convention to cut down on verbiage. (+ a b 3) is <u>not</u> a list; it is a string of characters representing one.

(set (quote .....) .....)

This is because far and away the most common use of the "bindings" of symbols is as the "values" of variables. So a special form is provided to allow us to set the values of variables more easily. It is called setq; we almost never use set: setq is usually what we want.

16) setq begins special forms. A setq form consists of "setq", a symbol name, and an inner form to be evaluated. The inner form is evaluated, and the binding of the symbol is made to be that value.

For example, instead of saying

(set (quote foo) (+ 4 bar))

we can say

(setq foo (+ 4 bar))

This is very much like an assignment statement in other languages, e.g.,

FOO := 4 + BAR ;

At this point, one might get the idea that dealing with Lisp consists primarily of handing forms to eval, and seeing what one gets back. In fact, this is the case. We are now ready to actually talk to Lisp on a real computer!

When we invoke Lisp on a computer, a whole new list-structure world is created beneath our fingertips. It already has a lot of useful and interesting objects in it, such as the subrs we have already described, and their name-symbols. It also has various other things, such as the obarray, and some chosen symbols like "nil".

Lisp continuously runs a loop: it calls "read" to get an object from the user, applies "eval" to it, and applies "print" to the result, thus showing the result of evaluating the user's form to the user. This "read-eval-print" loop is what Lisp does. Therefore, to find out what eval does with some form, you just type it in, and Lisp evaluates it and types back the result of so doing.

Now we are ready to run Lisp. On Multics, you start up Lisp by invoking the "lisp" command; that is, by typing

lisp

Lisp will respond with a "*", and await your typing in forms to be evaluated. On ITS, you start up a Lisp by typing

:LISP

Lisp will first respond by typing

```
LISP 1293 WITH WINNING NEW I/O
ALLOC?
```

to which you should answer negatively by typing an "N". Then it will type a "*" and await your typing in forms.

To get out of either, cause the "quit" subr to be applied to no arguments by typing "(quit)". On Multics, the "quit" command returns to command level; on ITS, it kills the "LISP" job, and returns to DDT.

17) quit takes no arguments, and brings about the end of the world.

The following is a dialogue with Multics LISP.

```
Multics 33.0: MIT, Cambridge, Mass.
Load = 17.0 out of 85.0 units: users = 17
login Weinreb SIPB
Password:

You are protected from preemption.
Weinreb SIPB logged in 01/06/78  2139.5 est Fri from ASCII terminal "none".
Last login 01/06/78  1801.7 est Fri from ASCII terminal "none".
No mail.
r 2139 3.743 64.024 1221

lisp    I INVOKE LISP.  IT TYPES "*" AND GOES INTO THE
*                READ-EVAL-PRINT  LOOP.
(set (quote a) 3) I TYPE THIS.
3               IT ANSWERS THIS.
I SKIP THIS SPACE DELIBERATELY.
(setq b 7)
7               SAME AS "(SET (QUOTE B) 7)".

a
3               EVAL APPLIED TO 'a' GIVES 3.

b
7

(+ a b)         APPLY "+" TO BINDINGS OF "a" AND "b".
12

(+ a b 2)
14

(setq list-1 (quote (a b c)))    THE BINDING OF "list-1" BECOMES
(a b c)                                                  "(a b c)"

(setq list-2 (quote (d e)))
(d e)

(cons list-1 list-2)  MAKES A NEW CONS.
((a b c) d e)

(setq x list-2)
(d e)

(rplaca list-2 (quote foo))  I CHANGE THE CAR OF THE CONS TO
(foo e)                      WHICH "list-2" IS BOUND.

x                                "x" WAS BOUND TO THIS CONS TOO.
(foo e)

(rplaca (cdr x (+ a b)))   WHOOPS!  AN ERROR — "cdr" GOT 2 ARGS,
lisp: wrong number of arguments - eval   ((cdr x (+ a b)) (nil . 1)) WANTED
                                                                     1.
```

```
;bkpt wrng-no-args
(ioc g)
Quit
*
(rplaca (cdr x) (+ a b))
(12)

x
(foo 12)

(setq ducks (quote (Huey Louie Dewey)))
(Huey Louie Dewey)

(car ducks)
Huey

(car (cdr ducks))
Louie

(rplaca (cdr ducks) (quote Louis))
(Louis Dewey)

ducks
(Huey Louis Dewey)

(quit)
r 2146 2.032 87.502 1988
```

"ioc" is an fsubr. "(ioc g)" goes back to the toplevel read-eval-print loop.

I CAN MODIFY OTHER CONSES, TOO.

A LIST.

THE FIRST ELEMENT OF THE LIST.

THE SECOND.

CHANGE THE LIST.

AND IT IS CHANGED.

LEAVE LISP.

Notes on the Programming Language LISP

by Bernard Greenberg

Part II

PART 2

A substantial simplification of life is provided by the reader for typing in those ever-so-useful forms whose car is "quote". The character "'" is recognized specially by the reader. Whenever the reader sees a "'", it reads in the printed representation of an object right after the "'", and produces a list whose first element is the symbol "quote", and whose second element is the object read in. For instance, instead of

```
(cons (quote a) (quote (b c d)))
```

we can write

```
(cons 'a '(b c d))
```

This is simply a shorthand notation which makes things easier to type in. There is no special type of object in Lisp corresponding to the "'" character; this is just a feature of the reader.

The character "'" is not a shorthand for the symbol "quote"; the following are equivalent:

'a	(quote a)
'((a b c))	(quote ((a b c)))
'quote	(quote quote)
''a	(quote (quote a))

PROPERTIES and PROPERTY LISTS

We now deal with the last fundamental constituent of a symbol, its property list. The property list of a symbol is a collection of pairs of of objects, which the Lisp programmer may use to attribute arbitrary and random properties to that which the symbol represents to him.

For instance, let us posit a symbol named "Fred", which, in some particular Lisp world, represents a fellow with that name. We wish to record that Fred is 33 years old, has blue eyes, and employs Faith, Hope, and Charity. We represent this in Lisp by giving the symbol named "Fred" an "age" property of a fixnum "33", an "eyes" property of a symbol named "blue", and an "employees" property of a list which prints as (Faith Hope Charity).

To say that the symbol named "Fred" has an "age" property of some Lisp object, in this case a fixnum of magnitude 33, means that two objects, being a symbol named "age" and this fixnum, are related in a very special way by a fundamental data structure associated with "Fred" called his property-list. The first object in this relationship is always a symbol, which is called the indicator, and the the second object is called the property. The indicator symbol, which should be thought of as a "property name", says which or what property, and the other object says what the value of that property is.

The beauty of property lists is that any symbol may have a random and indefinite collection of properties, whose indicators and values may not even be known at the time the program is written. The property list is one of Lisp's most powerful mechanisms for the accretion of arbitrary and extensible data.

The internal structure of the property list of a symbol is not very interesting. But if you care, it is list of all the indicators and properties this symbol has, indicator, property, indicator, property, indicator, property, and so on. Esoteric programs can deal with the property list itself, via the plist and setplist subrs, but this is rarely necessary. See the manual for more details. (1)

The way we normally manipulate properties is via three subrs, of unfortunate asymmetrical nomenclature, which we will learn

---

(1) Even though programs don't look at property lists directly all that often, the interactive user often looks at them, via forms like

```
(plist 'gruzzle)
```

to "find out" interesting "facts" about some symbol (in this case, one named "gruzzle").

about next. The first argument to each of them is always the symbol whose properties we wish to deal with.

18) get takes two arguments. The first is a symbol, the second a symbol to be used as an indicator. If the first argument has a property under the indicator of the second argument it is returned. Otherwise, nil is returned. Note that if the property happened to be nil, then one could not separate this from the case where the symbol had no property under the given indicator.

Example:

```
(get 'Fred 'employees)
```

evaluates to

```
(Faith Hope Charity)
```

19) putprop Applied to three arguments, a symbol, a property and an indicator. Gives the first argument (a symbol) a property with the third argument as the indicator (a symbol used as a property name) and the second argument as the property. The second argument is returned. Watch out here, read it again, for the order of arguments is counterintuitive.

20) remprop takes two arguments, a symbol and an indicator. Removes the property (and indicator) if it exists.

Hence, if we evaluated

```
(putprop x 'Fred 'father)
```

whatever symbol "x" is bound to gets a father property of "Fred".

If we then evaluated

```
(get (get x 'father) 'age)
```

we get 33.

Properties are the most common way to store changeable information in Lisp programs. Representing objects to be modeled by symbols and their attributes by properties is a better way to save and modify information than structured networks of conses for several reasons. Ease of programming and debugging is one reason, for symbols and properties have names. Secondly, new indicators can be added as you need them, without changing the entire program. The property list

is unstructured, and open-ended.

Notes on the Programming Language LISP

Earlier we learned how to construct, modify and inspect DATA OBJECTS in the list-structure world.

We have learned about a class of data objects, called FORMS, which specify action. We learned of eval, the subr which interprets forms. Also, we dealt with the interpreter and learned how to talk to Lisp.

Now we learn how to write programs.

Lisp is an applicative language -- All Lisp programs are expressed as collections of functions, which take objects as input and return objects as output. Subrs are a special case of functions. Subrs are functions written in machine-language which are (usually) part of the Lisp system. Programs are composed of user-supplied functions, which are written not in machine-language, but in Lisp. Such functions are composed of forms, which themselves consist of directions for the application of subrs and, perhaps, other user-supplied functions. One of the results of this is that all user-defined functions appear in forms just as the primitives of the language do. For example, a user might define a function "double" that would return twice its (fixnum) argument. This function would then be used exactly as though it were a subr; (double 3), when evaluated, would return 6.

Designing and constructing a Lisp program consists of creating and implementing a set of functions which deal with the problem domain in a useful and interesting fashion. Our next task is to learn to construct our own functions. Once we have learned this, there is little more to acheiving proficiency in Lisp than learning of the wide variety of available subrs and accumulated tricks of the trade.

Functions in Lisp are expressed by a miraculous and powerful naming operator called "lambda". Lambda is a way of generalizing a form. Lambda is a way of taking a form, which is an expression of a computation with specific quantities, and using it to express a computation with arbitrary quantities.

Let us proceed with the development of this doubling-function, as a first example of how functions are expressed in Lisp. We already know how to double specific things. For example, if we wanted to double 6, we would write the form

```
(+ 6 6)
```

On the other hand, if we had a symbol "v" which were bound to an object (which had better be a fixnum), and we wanted to compute that fixnum's double, we might write

```
(+ v v)
```

This is all well and good, but is only useful in the case that there happens to be a symbol named "v" lying around that happens to be bound to the fixnum whose double we wished to compute. Suppose, on the third hand, we had the result of some hideously complex machination, such as

```
(+ foo1 foo2 (* xbar yy17))
```

that we wished to double. We might just write

```
(+ (+ foo1 foo2 (* xbar yy17)) (+ foo1 foo2 (* xbar yy17)))
```

On the fourth hand, we'd rather not. Not only is this complex, error-prone and obfuscatory, but if the complex machination involved some application of a function which had side-effects, it would be outright wrong, for the side-effects would happen twice. Furthermore, computing the quantity twice is implied, and this is inefficient as well as logically erroneous. We don't want to compute this thing twice and add the two results. We want to say,

"Compute this here quantity. Then add it to itself."

We could do this by assigning this quantity to some variable, say, our friend "v", as follows:

```
(setq v (+ foo1 foo2 (* xbar yy17)))
(+ v v)
```

This is a lot better, but this has several problems. First of all, it requires us to either make sure that nobody else is using "v" for anything, or reserve a variable just for this purpose. Another good problem is that two forms are required: first a "setq" form and then a "+" form. To express a result to be used in a form, we need a single form. Two forms just won't do.

The form "(+ v v)" is appealing as an expression of the concept of adding something to itself. Its only real problem is that it is so specific; it deals only with a totally specific and almost always irrelevant symbol named "v". It says,

"Add the current binding of "v" to itself."

What we really want to say is:

"Add something to itself."

A good compromise might be:

"Call it "v". Add v to itself."

In other words, we want to let "v" be a name for the result of the thing which we want to double.

In Lisp, we might want to say,

"Let v be bound to the object of interest for the moment. Evaluate (+ v v)."

This is how we deal with the notion of function in Lisp. We write the above sentence in Lisp as

```
(lambda (v) (+ v v))
```

This Lisp fragment (it is not a form) is Lisp for either of the above indicated statements. This lambda expression is a function, just like a subr.

We can use a lambda expression in forms, to specify the function to be applied. Just as

```
(+ (* 7 (car x)) 5)
```

says, in some sense, "Do this with the object gotten by multiplying the car of x's binding and 7, and 5: add them",

```
((lambda (v) (+ v v)) 3)
```

says, "Do this with the fixnum object 3: Call it "v". Add v to itself." The lambda expression is a thing to do with objects, something which can be applied to objects, just like a subr. It says "I am a thing to do with an object, which I will refer to as "v". I will add v to itself." The result of applying this thing to a fixnum 3 is a fixnum 6.

The form

```
((lambda (v) (+ v v)) 3)
```

is called a lambda combination, which means that it is a form whose first element is a lambda expression. There are names for the parts of the lambda expression: the list "(v)" is called the lambda list; v is called a lambda variable. The form inside the lambda expression, "(+ v v)", is called the body.

Similarly, we can express functions of more than one argument in a natural way:

```
((lambda (x y) (* (+ x y) (- x y))) (+ 7 w) (- v 3))
```

This says, "Where x is the result of evaluating (+ 7 w), and y is the result of evaluating (- v 3), evaluate (* (+ x y) (- x y))".

Eval recognizes a lambda expression, i.e., a list whose car is the "chosen" symbol lambda, as a representation of a function when it appears in the car of a form. When eval is given a form which has a lambda expression as its car, the other elements of the list are evaluated to produce the objects to which to apply the function, just as it does for a subr.

Eval somehow must accomplish the "call it v" stuff, so that it may just evaluate the "(+ v v)" in its normal fashion and get the right answer. "Call it v" can be accomplished as follows:

1. Memorize what v's current binding is; put it away someplace.
2. Make the binding of v be the object desired.

And after evaluating the form "(+ v v)",

3. Recall what the binding of v was, and restore the binding of v to be that.

This sequence of three operations is called binding v to the object desired. It has the effect of not disturbing whatever use was being made of v at the time. It makes it appear as though v were like a "local variable" or "temporary variable" in that form. This is highly desirable.

Note that binding v means the saving and restoring of v's binding as well as giving v a new binding. When you simply change v's binding without remembering the old one, this is called setting v, because it is what the "set" subr does. Binding is a temporary, reversible operation.

Understand that "lambda" is not the name-symbol of any subr; lambda expressions are not forms, and are not intended to be evaluated. A lambda expression is a representation of a function recognized as such by eval. Lambda expressions are very much like subrs: they are verbs. A lambda combination is a perfectly good form, and is meant to be evaluated; it directs eval to apply a lambda expression to some arguments, which are obtained by evaluating the remaining elements of the form.

In summary, here is how eval evaluates a lambda combination form:

1. Evaluate the second through nth elements of the form.
2. Apply the lambda expression to the results of step 1.
   2a. Bind each of the variables of the lambda expression

to the corresponding object obtained from step 1, being careful to save the old bindings.
2b. Evaluate the body of the lambda expression, obtaining an object as a result.
2c. Restore the old bindings of the lambda variables.
2d. Return the result of step 2b.

Subrs and lambda expressions are both specifications of a thing to do with some objects, i.e., functions. When we want to use a subr, we have its name-symbol as a way of specifying to eval a desire to use that subr. The next logical step is to somehow provide name-symbols for lambda expressions.

Sure enough, there is a way to connect a lambda expression to a symbol, just as a subr is connected to its name-symbol. For instance, we can connect the lambda expression "(lambda (v) (+ v v))" to the symbol "double". Once we have done this, we can say

```
(double 3)
```

or

```
(double (+ 3 (* 6 5) (- 2 1)))
```

and eval will dutifully use that lambda expression as though we had said

```
((lambda (v) (+ v v)) 3)
```

or

```
((lambda (v) (+ v v)) (+ 3 (* 6 5) (- 2 1)))
```

When we have done this, we have defined our own function. We can use its name-symbol just like the name-symbol for a subr.

We define functions by means of the fsubr called "defun". Here is how we would define our doubling function:

```
(defun double (v) (+ v v))
```

Note that "defun" is an fsubr! The symbol "double" will not be evaluated when the "defun" form is evaluated, nor will the "(v)" nor any other part of it. Instead, "defun" fsubr will create a lambda expression out of the above, and attach it to the symbol "double".

19) defun, an fsubr, takes a symbol, a lambda list, and a body. It creates a lambda expression from the lambda list and body, and places it on the property list of the symbol in such a way that

eval will know that that symbol is the name-symbol for that lambda expression.

Here are some more examples of function definitions:

```
(defun addone (x) (+ x 1))
```

The "addone" function will take a fixnum and return a fixnum whose magnitude is one greater than its argument's magnitude. That is, it will add one to something.

```
(defun recons (x y) (cons (car x) (cdr y)))
```

The "recons" function takes two conses, and returns a new cons whose car is the car of its first argument and whose cdr is the cdr of its second argument. For example, the result of evaluating

```
(recons '(a . b) '(c . d))
```

would be

```
(a . d)
```

```
(defun list-of-three-things (a b c)
       (cons a (cons b (cons c 'nil))))
```

The "list-of-three-things" function takes three arguments, and returns a list of three elements, being the three arguments to which it was applied. For example,

```
(list-of-three-things 3 '(a . b) 'fred)
```

would evaluate to

```
(3 (a . b) fred)
```

++

Congratulations! You have managed to make your way through enough of this language to be able to write your own functions.

[[[[[[[[[[[[[[[[[[[[[[[[[[[[]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]

Now that we are able to define functions, we need a complete programming language. This requires things like conditionals, loops, recursion and the ability to group imperatives together to form a program.

Let's start with conditionals and predicates.

There is an fsubr known as "cond", to be discussed shortly, which conditionally evaluates forms and returns different values. This is used to make decisions. It is similar to the "if" constructs of FORTRAN, ALGOL, and PL/I.

In order to make decisions we need predicates. These are subrs (or functions of your own construction) which return an indication of truth or falsehood. In Lisp, falsehood is represented by the symbol "nil". Truth is represented by any object other than "nil"; conventionally, the symbol "t" is used for this purpose. Predicates are the interrogatives of Lisp; they are used to ask questions.

Here are some useful predicates:

21) eq — Applied to any two objects, returns "t" if they are the same object, otherwise it returns "nil".

22) < — Applied to two arguments, which must be fixnums, and returns "t" if the first is numerically less than the second, otherwise it returns the chosen "nil".

23) > — Like <, but it returns "t" if the first argument is greater than the second.

24) not — Returns "t" if the argument is "nil", else it returns "nil".

25) fixp — Returns "t" if the argument is a fixnum, else "nil".

26) = — Applied to two fixnums. Returns "t" if they are numerically equal, "nil" otherwise.

27) symbolp — Returns "t" if the argument is a symbol, otherwise "nil".

28) atom — Returns "nil" if the argument is a cons, otherwise returns "t".

To make passing around objects which signify truth or falsity easier, the chosen symbols "t" and "nil" are self-evaluating: they are always bound to themselves.
So:

```
(setq x 'nil) <=> (setq x nil)
```

Lisp provides us with cond. cond is an fsubr. It is not easy to grasp at first sight. Recall that the arguments in a form whose car is an fsubr are not evaluated by eval before the fsubr is invoked. Rather the fsubr itself may call eval if it desires. A form

using cond looks like this:

```
(cond (p1 c1a c1b c1c ... c1x)
      (p2 c2a c2b c2c ... c2x)
          . . . .
          . . . .
      (pn cna cnb cnc ... cnx))
```

(p = predicate form, c= consequent form)

where all of the p's and c's are forms. Each (p c .....) is called a cond clause.

Here is an example of a cons form with three cond clauses.

```
(cond ((eq man 'Max) 3)
      ((eq woman 'Sarah) (+ 2 3))
      (t (print 'foo) 7))
```

If the symbol "man" is bound to "Max" (note the quote in the cond clause), 3 is returned (remember numbers are self-evaluating). If "man" is not bound to "Max", but "woman" is bound to "Sarah", 5 is returned. If neither of the above are true, "foo" is printed, and 7 is returned (recall that the symbol "t" is always bound to itself, and so is not "nil").

In terms of the general form above, cond works as follows:

> cond evaluates p1. If the result is not "nil", i.e., if it represents truth, cond evaluates c1a, c1b, ... in succession. The value of the last c is the value returned by cond. There can be any number of c's in each clause. If p1 evaluates to "nil" then p2 is evaluated, and so on until it finds some p that does not evaluate to nil. That is, it searches the cond clauses for one whose predicate is true, so to speak, and when it finds such a clause, it evaluates all of the consequents. If it cannot find such a clause (if all the predicates evaluate to "nil"), it returns "nil" itself.

Note that this conditional evaluation facility not only provides conditional flow of control, but conditional selection of values as well.

The PROG Story

Lisp also gives us a FORTRANesque (or Algol or PL/1-esque if you prefer) facility for evaluating several forms in sequence and throwing away their values. The basic one comes in three flavors: prog2, progn, prog. Obviously, if you had a function to evaluate two forms where before you could only have one (although the number of places in MACLISP where this is true has been minimized) you could in principle make ever expanding trees of forms.

Thus we have prog2. prog2 is used to cause two forms to be evaluated in sequence. It returns the value of the second of the forms. For example, if we give eval

```
(prog2 (setq x (+ 5 2))
       (print 'bar))
```

first the "(setq x (+ 5 2))" gets evaluated, making x be bound to 7, and then the "(print 'bar)" is evaluated, printing "bar" and returning the symbol "t" (print always returns "t"), and so prog2 returns t. (1)

So, we could have

```
(prog2 (prog2 (prog2 (setq x (+ y 3))
                     (setq y 27))
              (prog2 (setq z (read))
                     (print (* x y z))))
       (prog2 . . .
```

and so on.

We obviously could build up programs in this way, but this is clearly inelegant. So we have progn, which is like prog2 but causes the sequential evaluation of any number of forms. As above, eval does all the work by evaluating the arguments to progn:

```
(progn (print 'Type-in-two-numbers)
       (setq x (read))
       (setq y (read))
       (print 'answer-is)
       (print (+ x y)))
```

Is very FORTRAN.

---

(1) prog2 is not actually an fsubr; it is simply a subr which returns its second argument. The act of evaluating the form in which "prog2" appears causes these forms to be evaluated; all prog2 need do is return the second argument. If this confuses you, forget about it; it is a hack.

The most general member of the family is the prog, which allows "labels" (which are like PL/I labels or FORTRAN statement numbers) in such constructions as above. prog is an fsubr, and therefore eval does not evaluate the elements of the cdr of the prog's form before applying prog to it. prog, upon getting the form, evaluates all elements of its form except symbols. (Obviously, evaluating a symbol and then throwing away the resulting value is not very interesting anyway.) The symbols are like "labels" or "statement numbers"; they mark a place in the prog body which can be "gone to".

To make it all jell, there are two odd functions, go and return, to use within progs.

29) go — A go form looks like (go label). go is an fsubr. It conspires with prog in a strange way. Everything then stops what it is doing, and prog continues evaluating the elements of its form at the point where the label appears.

For example:

```
(prog (x)
      (setq x 0)
   a  (setq x (+ 1 x))
      (go a))
```

adds up a lot of ones.

The first thing in the form after the prog is somewhat peculiar. It is not a form to be evaluated, but rather a possibly empty list of temporary "variables" (symbols) to be used in this prog. It is the same as if they were lambda variables of a lambda expression, and Lisp binds them to nil when the prog form is evaluated. Remember that when you bind a symbol, you remember its previous binding; it is restored at the end of the evaluation of the prog. Thus, these symbols may be used as variables within the prog without fear of disturbing other use of them.

30) return is a subr which causes prog to be exited with return's argument as a value. That is, the "prog" returns the object which was the argument to "return". If a prog gets to the end of the elements of its form, it returns nil.

Now we will contruct a simple function using prog to compute Fibonacci numbers. Fibonacci numbers you will recall, are the numbers:

1, 1, 2, 3, 5, 8, 13, 21, 34, . . .

where each is the sum of the proceeding two. This function fib0 computes the Nth Fibonacci number in the sequence. This program should be obvious, with one exception.

Since prog2 was so useless in light of progn, a most interesting and curious use was found for it. prog2 actually accepts any number (greater than 1) of arguments, but it always returns the second one!! Hence one can exchange the values of two variables, i.e.:

```
t=x
x=y
y=t
```

(in FORTRAN or BASIC)

which in Lisp can be done as follows:

```
(setq y (prog2 0 x (setq x y)))
```

The zero is evaluated and then thrown away.

```
(defun fib0 (x)
   (prog (old older)
      (setq old 1)
      (setq older 1)
   a  (cond ((< x 2) (return old)))
      (setq older
            (prog2 0 old
                   (setq old (+ old older))))
      (setq x (- x 1))
      (go a)))
```

@@@

One particularly neat pair of Lisp constructs is the pair of control-flow brothers and and or. Both of these fsubrs, De Morgan's Law duals, evaluate the elements of their form until some argument evaluates to "nil" in the case of and, or non-nil for or. The value of the last form evaluated is returned. One can use them for logical constructs, like

```
(and (> x 5) (< x 15))
```

to test if x > 5 and x < 15 (octal). Hence, they can be used to compound forms into a compounded logical expression. Well, this is very common in all programming languages. However, here the similarity ends. The neatest thing about "and" and "or" is that their evaluation order is strictly defined, and conditional. Suppose that we have no idea to what x is bound. We want to know if x is bound to a cons whose car is the symbol "foo". The form

```
(and (not (atom x))
     (eq (car x) 'foo))
```

will provide the answer. The point of this is that in general, it is illegal to ask for the car of a symbol. The "and" form above stops evaluating if x is bound to a symbol, and it never tries to take the car of the symbol. As a matter of fact, it is very common is MACLISP to see "and" and "or" used as a substitute for cond in simple cases, e.g.,

```
(and (= x 7) (go moe))
```

has two less parentheses than

```
(cond ((= x 7) (go moe)))
```

and since there are fewer parentheses, it is easier to see what is happening.

```
(or (symbolp x) (print 'x-is-not-a-symbol))
```

is a very graphic statement of "unless x is a symbol, print out so and so".

- -

Most programming languages have iteration, and many support recursion as well. Recursion in LISP is very cheap and thus almost de rigeur. The natural recursive definitions of lists and list structure lend themselves to recursive processing. (Recursion is when a function can call itself.)

A recursive (and very natural) rewrite of the Fibonacci program is as follows:

```
(defun fib1 (x)
    (cond ((< x 3) 1)
          (t (+ (fib1 (- x 1))
                (fib1 (- x 2))))))
```

Note that in the first recursive call to fib1, "x" will be bound to the current "x" minus "1", as the recursive invocation is executed. The binding of "x" will then be restored so that the call to fib1 on the next line indeed refers to the same "x" as the previous one.

(1)

---

(1) This, however, is an extremely slow program, taking time proportional to exp(x), because this particular algorithm for computing Fibonacci numbers computes the same values many times. It is not recursion in Lisp which is slow, but this particular algorithm.

We have written two versions of the Fibonacci function: "fib0" was iterative, and "fib1" was recursive. Here is a function that cannot be written iteratively, but whose recursive definition is natural, simple, elegant, and efficient. It prints out the printed representation of all of the non-cons objects in a piece of list-structure. For example, given

```
((a b) (c (d e) 6) f)
```

it will print

```
a
b
nil
c
d
e
nil
6
nil
f
```

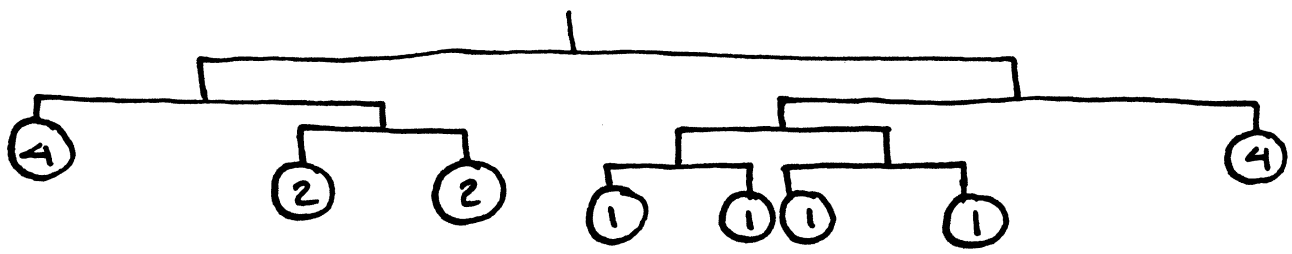
Here it is:

```
(defun fringe-print (x)
    (cond ((atom x) (print x))
          (t (fringe-print (car x))
             (fringe-print (cdr x)))))
```

Note the appearance of "nil" in several places. If you draw out the list structure corresponding to the argument we gave fringe-print, you will see why.

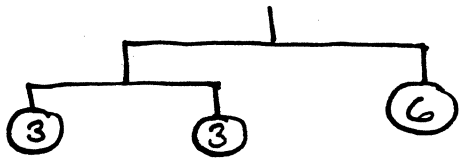
Another typical use of recursion, dealing with recursive list structure is as follows:

We have a representation of a mobile (a hanging ornament) with weights as nodes. Such a mobile might look like



(A mobile is either a single weight, like,

or two legs which are mobiles themselves, such as,



)

We represent it in list structure as:

```
((4 (2 2)) (((1 1) (1 1)) 4))
```

We wish to know if such a mobile is balanced. A mobile is balanced if and only if either it is just a single weight or both its legs are balanced and of equal weight. The weight of a single weight is whatever it is, the weight of anything else is the sum of the weights of its legs. The following two functions perform this task: (1)

```
(defun weight (x)
    (cond ((atom x) x)
          (t (+ (weight (car x))
                (weight (cadr x))))))

(defun balancedp (x)
    (cond ((atom x) t)
          (t (and (balancedp (car x))
                  (balancedp (cadr x))
                  (= (weight (car x))
                     (weight (cadr x)))))))
```

"balancedp" is applied to a representation of a mobile and returns "t" if the mobile is balanced, "nil" otherwise. Thus, it is a predicate.

---

(1) cadr is a subr which obtains the car of the cons which is the cdr of its argument (the CAr of its cDR). That is, "(cadr x)" is the same as "(car (cdr x))". It is one of a family of subrs with names like caddr, cdar, cdadr, etc. being some other members. MACLISP supports all combinations up to caaaar and cddddr. Note that car applied to a list gives the first element, cadr the second, caddr the third, etc.

"weight" is an auxiliary function which is applied to a representation of a mobile to obtain its weight. Both functions are recursive, as the definition of both the mobile and its weight are recursive. The power of Lisp for this lies not in the ability to write a recursive program in it, but to represent recursively defined structure.

Although recursion in Lisp is common and very cheap, one must not succumb to the temptation to use it as a substitute for iteration where the latter is the obvious choice. It has been somewhat traditional, however, in presenting Lisp, to do precisely that. Among the applications of recursion where it is the right solution are the parsing, semanticating, and code-generating of a context-free or block-structured language, the solution of mathematical problems which are recursively defined, or, in general, any kind of thing which might get itself involved with itself recursively several levels down. Anything which recurses to do the same thing with the rest of something, such as doing a certain thing to every element of a list, probably ought to be iterating.

Lisp provides several mechanisms for organized iteration -- a conglomeration of cond's and go's constitutes disorganized iteration. The most common form of iteration in Maclisp is the do loop, an old friend to those who started out with FORTRAN. The actual variant of do used in Maclisp is a direct parallel of PL/1's do-repeat-while. One says,

```
(do variable initial-value repeat-value
    stop-test
    first-thing
    second-thing
    .
    .
    last-thing)
```

in general. For instance, one might say

```
(do i 1 (+ 1 i)
    (> i 100)
    (print (cons i (* i i))))
```

to print out all the numbers from 1 to 100 (octal) and their squares (very FORTRANesque application, note.) In this example, the variable is "i", the initial-value is "1", the repeat-value is "(+ i 1)", and the stop-test is "(> i 100)".

An exact definition of this kind of do is as follows:

Using the example above,

```
((lambda (variable)
         (prog ()
Rumpelstiltskin (cond (stop-test) (return nil))
                first-thing
                second-thing
                .
                .
                last-thing
                (setq variable repeat-value)
                (go Rumpelstiltskin)))
 initial-value)
```

In the above definition, "Rumpelstiltskin" is a label whose name is invisible to the programmer. This is really a definition, which is to say, that if you use a "return", (i.e., (return foo)) in a do, that will be the return value of the whole do. Note that one can also use labels and go's in a do (the stuff first-thing, second-thing,..last-thing, is called the body of the do), as it is really an elaboration upon a prog. We call the do's internal label "Rumplestiltskin" because you can't guess its name, and so it will never conflict with any of your own labels. This kind of thing where one function is just a way of saying a whole bunch of other forms made up out of stuff in the original form is called a macro, and we will learn about them in more detail, like how to make your own, later on. For now, observe that, in terms of what we know up to this point, do must be an fsubr, for its form is funny, i.e., eval does not simply evaluate the elements of the form after "do" as arguments, and hand them on to a machine language function. The "do" fsubr has complete control over the evaluation of stuff in that form, as it has to simulate all that lambda-setq-cond-prog-setq-go hackery.

We have just encountered the simpler of two forms of do, which is the older of the two. For many simple applications it suffices, but one can usually do something clever to do its task in a simpler way. The new form of do, however, is somewhat baroque, but more than makes up for it by its power. Instead of having one variable, the new "do" allows any number of variables! The neatest thing about it, however, is that all the repeat values are assigned in parallel, i.e., all the repeat values are evaluated before they are assigned to their respective variables. You can envision this as being done either with a whole bunch of temporary variables, or with the prog2 trick we learned about above. The new do has the syntax

```
(do variable-specs
    end-clause
    first-thing
    second-thing
    .
    .
    last-thing)
```

The body is identical to the older do, and can contain

labels and returns. The end clause is a list of several forms -- the first is a stop-test as above, and the second through last, if present, are forms to be evaluated when the end-test succeeds. If nothing in the body of the do has evaluated a return, the value of the do, when the end-test succeeds, is the last of these "exit forms".

The variable-specs is a list of many variable-spec's. If it is empty, i.e., none of them, this do simply does, until the end-test is met. Otherwise, a variable-spec is one of three options:

```
(var)
(var init)
(var init repeat)
```

In the first case, var is set at the top of the loop to nil, and never reset except by stuff in the body of the do. In the second case, var is set to the value of "init" at the top of the loop, and never set to anything else except by stuff in the body. In the last case, var is set to the value of "init" at the top of the loop, and set to the value of "repeat", reevaluated each time, at the bottom of the loop, where all assignments are made in parallel.

Here is still another revision of our Fibonacci number program which uses the new do format. Note how we take explicit advantage of the parallel assignment in this program. Note also that the body of the do is empty: some of the finest do's have no bodies, as the power is all in the repeat clauses. This is the most efficient version of the program so far.

```
(defun fib2 (x)
     (do ((ct x (- ct 1))
          (old 1 (+ old older))
          (older 1 old))
         ((< ct 3) old)))
```

The expansion of the newer form of do as a prog (such as the expansion given for the older form above in terms of lambda, etc.) is non-obvious, and is left as an exercise for the interested reader.

You should be able to figure out that the "do" fsubr figures out whether it has an old or a new format do by looking at the first element of its form: if that element is a symbol other than nil, the form is an old-style do; otherwise, it is a new-style do. Note that only fsubrs (like cond, prog, do, setq) have syntaxes, i.e., explicit rules how their forms have to be laid out (First comes a list of all the soandsos, and then a form which is evaluated every third time to test if the whatsit... etc.). Fsubrs, in general, are the control-flow constructs of the language. quote and setq, which must not be left out in any consideration of fsubrs, are basic artifacts of the evaluator. There are a few fsubrs (status and signp, for example) which really

have no claim to any such exalted status. Reconsider them at some other time.

Lists as Sets

The notion of a list, as encountered in previous lessons, is beholden at least to the printer and the evaluator. To the evaluator lists are the expression of forms: the first element of a list is a function, the remaining elements are forms to be evaluated to supply the arguments to which the function is to be applied. The printer prints lists by printing all the elements in order between a pair of parentheses.

Lists are a very useful notion. In everyday life, we deal with grocery lists, laundry lists, blacklists, other kinds of lists. These lists are not lists at all, in the computer sense, but rather expressions of sets of groceries, shirts, suspected Communists, and whatevers. Lisp lists are a very useful expression of sets, just as the cdr of a form is a set of forms for evaluation. Thus,

```
(Khrushchev Malenkov Bulganin Beria)
```

might be a list representing certain individuals singled out for some purpose.

```
((fish trout halibut mackerel)
 (amphibian frog toad salamander)
 (reptile crocodile lizard stegosaurus)
 (bird heron jay robin bluebird)
 (mammal bear cat dog student))
```

might be a set of classes of the vertebrate phylum, where each family is represented as a cons of its name and a list of members that we know about for some reason.

It is at dealing with precisely this sort of thing that Lisp excels: several functions are provided for dealing with lists as sets, and iterating over them. As a matter of fact, whenever a do loop is looping over a list, possibly one should be using one of these useful subrs instead.

They all expect their second argument to be a list. Remember that "nil" is a list, of zero elements.

15) cons of a thing and a list. You surely remember cons! (We saw it before, in chapter 1). Well, viewed as a list operator, one can see that cons creates a new list, with the thing supplied at the head, and the old list as the rest of the list. For example,

```
(cons 'a '(b c d))
```

results in a new list of four elements:

(a b c d)

30) memq of any object and a list. As a predicate, tells you if the object supplied is a member of that list (i.e., if it is eq to any element of the list). If not, "nil" is returned. If so, the cons whose car was the object supplied (Such a cons had to be part of the list by hypothesis) is returned, and since no cons can be "nil" ("nil" is a symbol, not a cons!!!), cond, and, or, etc., will take this as signifying "true". Hence, if x is bound to the list of Soviet diplomats above,

(memq 'Stalin x) gives

nil

(memq 'Bulganin x) gives

(Bulganin Beria)

(1)

31) delq of an object and a list. Takes the item out of the list, by rplacding the cons before whichever cons has the object as its car with the cdr of the latter. If the object was the first element of the list, this is obviously impossible, and delq hands you back the second cons of the list, hoping that you will use this for whatever purpose you had been using the other idea of the list for. In all cases, delq hands back the original list, suitably bashed. For example, using the list above,

(delq 'Bulganin x) gives

(Khrushchev Malenkov Beria)

and (delq 'Khrushchev x) gives

(Malenkov Bulganin Beria)

32) mapcar is one of a set of very powerful "mapping" functions. The first argument is a function, e.g., a name-symbol of a subr or a user-defined

(1) Note that the name "memq" (and "delq", which we are about to describe) has no connection with the name "setq": The name "setq" alludes to the fact that it is an fsubr like quote. The names "memq" and "delq" allude to the fact that they search down their lists applying the "eq".

function, or a lambda expression. The second argument is a list. mapcar walks down the list, applying the function to every element of the list in order, and produces a new list (not destroying the old), where every element is the result of the application of the function to the corresponding element of the original list. For example, suppose y is be bound to the zoological list above:

```
(mapcar 'car y)
```

returns (fish amphibian reptile bird mammal), a list of all the family names. It did this by applying car to each element of the list y.

33) mapc is like mapcar, but constructs no list. The second argument verbatim is mapc's value. That is to say, the applications are performed only for what side effects they might have. Again, assuming y bound to that zoological list,

```
(mapc '(lambda (z) (and (memq 'student (cdr z))
                        (print (car z))))
      y)
```

prints out the family name of the family that contains "student".

34) list of any number of arguments. list constructs a list out of the items to which it is applied. The first argument becomes the first element of the list. For example:

```
(list 'this 'that '(the other thing))
```

gives

```
(this that (the other thing))
```

Note that for three arguments,

```
(list a b c)
```

is equivalent to

```
(cons a (cons b (cons c nil)))
```

and so forth for any specific number of arguments.

35) append of any number of lists. append returns a new list, whose element are all of the elements of its arguments. For example:

(append '(a b c) '(d e f) nil '(g) '(h i)) gives

(a b c d e f g h i)

All of the conses of the created list are newly created, except for those of the last argument.

Notes on the Programming Language LISP

by Bernard Greenberg

Part III

PART 3

```
(defun simp (x)
    (cond ((atom x) x)
          ((eq (car x) '+)
           (simplify-plus (simp (cadr x)) (simp (caddr x))))
          ((eq (car x) '*)
           (simplify-times (simp (cadr x)) (simp (caddr x))))
          (t x)))

(defun simplify-plus (e1 e2)
    (cond ((and (fixp e1) (fixp e2))
           (+ e1 e2))
          ((and (fixp e1) (= e1 0))
           e2)
          ((and (fixp e2) (= e2 0))
           e1)
          (t (list '+ e1 e2))))

(defun simplify-times (e1 e2)
    (cond ((and (fixp e1) (fixp e2))
           (* e1 e2))
          ((and (fixp e1) (= e1 0))
           0)
          ((and (fixp e2) (= e2 0))
           0)
          ((and (fixp e1) (= e1 1))
           e2)
          ((and (fixp e2) (= e2 1))
           e1)
          (t (list '* e1 e2))))
```

A Simple Program

For your entertainment and education, we present two viable Lisp programs in this chapter. The first, the simpler of the two, is a simplifier of algebraic expressions, which could easily be part of a symbolic mathematical system, or a programming language interpreter or compiler. The second program is a game-playing program, which you will probably find somewhat amusing to play with.

The algebraic simplifier operates upon calculations, expressed as, of all things, Lisp forms. Such a calculation might be represented by the list structure which prints as

```
(+ (* u 0) (* 6 (+ c 0)))
```

This particular simplifier is rather limited; it deals only with expressions of addition and multiplication, of two addends or factors. The simplifications that it is capable of performing are:

1) Eliminating additions to zero, replacing them by the thing being added to zero, for example,

```
(+ v 0)
```

is simplified to

```
v
```

2) Eliminating multiplications by 1 in a similar way.

3) Eliminating multiplications by 0 by a 0, for example,

```
(* f 0)
```

is simplified to

```
0
```

4) Reducing calculations on two constant operands to a constant, for example

```
(+ 3 4)
```

is simplified to

```
7
```

Although all of the above cases are quite easy to deal with, we want to be able to simplify calculations built up of littler

calculations built up of littler calculations yet in a perfectly general manner. For example, we want

```
(+ (* u 0) (* 6 (+ c 0)))
```

to be simplified to

```
(* 6 c)
```

Thus, we want our simplifier to do its job recursively, to work on subexpressions down to the utmost level in an identical manner, and to benefit by having simplified the constituents of any expression before dealing with that expression on its own merits.

To do this, we break up the task of simplification into two components:

1) Being recursive and figuring out what's going on

and

2) Simplifying the actual expressions, with all issues of recursion having been dealt with already.

The program itself is made up of three functions, defined by three defun forms. The names of the functions are simp, simplify-plus, and simplify-times.

The implementation of the first task is provided in the function simp, which is, by the way, the top-level function of this program, i.e., the function that we actually apply to the structure we wish to simplify. simp looks at the form he is handed, and does different things depending on what it looks like. A symbol or a fixnum here represents a variable or a number, respectively. In and of themselves, the quantities represented by these objects cannot be simplified any further, and simp returns them as is. If handed a list, however, simp knows that a calculation is represented, and he passes the buck to one of his friends who can deal with the specifics of addition or multiplication as appropriate. Before doing so, however, simp calls himself recursively (or recurses) upon the operands of the expression, simplifying them for all they are worth. It is these processed operands that are handed to the simplifiers of addition and multiplication, simplify-plus and simplify-times.

The strategy of recursing upon pieces of one's input argument before dispatching work to others, giving them the results of these recursions, is an important and widely used technique in Lisp programming. eval himself is like this; he recurses upon the argument-forms in a form before dispatching the work to some subr, passing the latter the result of these recursions (which are the arguments to the subr)!

The task of simplify-plus and simplify-times is thus simplified, so to speak, by what simp has already done before calling them. The arguments passed to these two functions are always simplified to the best of the ability of the program.

We will examine in close detail how simplify-plus works, since simplify-times is very similar to it. simplify-plus knows about two kinds of simplification: he knows that addition of something to zero is just the thing itself, and that the sum of two constants is just another constant. So what simplify-plus will do depends on whether its arguments fit any of these specifications.

To take action conditional on the argument, a cond form is used. It has four clauses. The predicate form of the first clause is

```
(and (fixp e1) (fixp e2))
```

This will be true (i.e. evaluate to something other than nil) if both of the forms

```
(fixp e1)        and        (fixp e2)
```

are true; that is, if both e1 and e2 are fixnums. If they are, the predicate is true, and so the consequents are evaluated. In this clause, there is only one consequent: (+ e1 e2). So the two fixnum arguments are added, and simplify-plus returns their sum.

If the predicate of the first clause is not true, then the next clause is tried; that is, simplify-plus continues to search for a case it knows how to simplify. What the next clause says is: "If the first argument is a fixnum whose magnitude is zero, then return the second argument.". Notice that before evaluating the (= e1 0), it must first make sure that e1 is a fixnum, because the "=" subr only takes fixnums (if you give it anything else, an error will be caused). Similarly, the third clause checks to see if the second argument is a fixnum 0, and, if so, it returns the first argument.

If none of the first three clauses has a true predicate, simplify-plus couldn't find anything to simplify, and so it goes to the fourth and final clause of the cond. The predicate of this clause is just the symbol t, so its consequents will always be evaluated if none of the first three forms' predicates are. It is a last resort: it simply returns an expression which represents the addition of its arguments, that is, a list of three elements: the symbol "+", the first argument, and the second argument.

Note carefully the difference between the first and fouth clause. The first clause actually performs the addition itself; the fourth does not do any addition, but returns a form which represents an addition.

simplify-times is very similar to simplify-plus; it checks for two fixnum arguments, one argument's being zero, and one argument's being one. It also has a last resort at the end of its cond, in which it returns a representation of a multiplication.

Here is an example of what the program does; suppose we applied simp to a list which looked like

```
(* (+ a 0) (* 2 4))
```

First, simp would discover that his argument was a list whose car is the symbol "*" (the third clause of simp's cond) and would call simp on the arguments; first is would call simp on

```
(+ a 0)
```

simp would see this, and call simp on the symbol a, which would return a. Next it would call simp on the fixnum 0, which would likewise return 0. Finally it would call simplify-plus on the results of the two recursive calls to simp, that is, on a and 0. simplify-plus's third clause would take action, and return a.

This would then get returned by simp. Next, the first of all of these simps would call simp again on the list (* 2 4). After performing boring simplification of 2 and 4, yielding just 2 and 4, simplify-times would be called in. Its first cond clause would do the job, multiplying 2 by 4 to give 8 (decimal). It would return 8 back to the top simp.

Finally the top simp would call simplify-times on the two returned values of the recursive calls to simp, namely, a and 8. simplify-times does not know how to simplify themultiplication of a and 8, so it will create and return a list which looks like (* a 8). This is what the top-level call to simp will return; it is the final answer.

In case that mess of recursive calls is unclear, here is a picture of what would happen: each line of the picture represents the application of one of the three functions, and the lines are arranged in chronological order. The indentation points up who is calling whom.

```
simp (* (+ a 0) (* 2 4))
 simp (+ a 0)
  simp a
  simp 0
  simplify-plus a 0
 simp (* 2 4)
  simp 2
  simp 4
  simplify-times 2 4
 simplify-times a 8
```

The important thing about this whole program is that it is not manipulating numbers the way typical FORTRAN, BASIC, or ALGOL programs do; it is working with algebraic expressions. Although what it does is very simple (it was only written to help teach Lisp), it is hoped that the student can see where this sort of thing could lead. The program could be improved to know about other mathematical functions, and how to handle any number of arguments; it could try to combine common subexpressions, factor things, expand things, and so on.

You might also imagine other programs to differentiate and integrate expressions, producing other expressions, or solve equations symbolically. In fact there is a very, very large Lisp program called MACSYMA which does all of these things, and it is used every day by mathematicians and physicists. It includes a wide variety of sophisticated simplifying functions.

So for all its simple-mindedness, the program above is really a typical application of Lisp, and such manipulation of structured information is one of the things at which Lisp excels.

==========

A More Complex Program

We now present another example of a usable program: this time, a game-playing program. In addition to being fun to use, this program gives some taste of using Lisp for language processing, and deals with a minimal sampling of artificial intelligence. Here is a transcript of a conversation with this program, including the user's invocation of the Lisp subsystem on Multics, and Multics's ready message (job completion indication) when the game is over. All of the questions are asked by the program. The terse responses are those of the game-playing user.

```
lisp animal
Let's play a game. Choose a random animal.
I'm gonna try to guess it by asking you questions,
and you give me yes-or-no answers. OK? let's go.
Does it have horns?
yes
Is it a buffalo?
no
Well, I'm not too sharp today. I give up.
Just what kind of beast did you have in mind?
a gazelle.
Tell me something about a gazelle which is not true about a buffalo.
a gazelle is graceful
That was fun. Wanna try again?
maybe
```

```
Hey, can you give me a yes or no answer?
yes
Does it have horns?
no
Is it a butterfly?
no
Well, I'm not too sharp today. I give up.
Just what kind of beast did you have in mind?
a pig
Tell me something about a pig which is not true about a butterfly.
a pig is slovenly
That was fun. Wanna try again?
yes
Does it have horns?
no
Is this animal slovenly?
yes
Is it a pig?
yes
Hey hey I sure am clever, huh?
Lisp MUST be a great language.
That was fun. Wanna try again?
yes
Does it have horns?
yes
Is this animal graceful?
you are not graceful
Hey, can you give me a yes or no answer?
no
Is it a buffalo?
no
Well, I'm not too sharp today. I give up.
Just what kind of beast did you have in mind?
bull
Tell me something about a bull which is not true about a buffalo.
you tell me something you ttl moron
Aw, be serious. I asked you a real question.
it would marry a cow
That was fun. Wanna try again?
yes
Does it have horns?
yes
Is this animal graceful?
no
Is it so that it would marry a cow?
yes
Is it a bull?
yes
Hey hey I sure am clever, huh?
Lisp MUST be a great language.
That was fun. Wanna try again?
no
```

```
Do you want to save me?
yes
Type:   lisp animal
to play this game again.
r 1905 3.573 88.624 1162
```

The program maintains a data-base representing a series of questions whose answers select some specific animal. The game-playing user is asked to secretly choose an animal. The program issues a series of questions, narrowing down the possibilities at each answer, until there is only one animal that the program knows about that meets all these criteria. At that point, the user is asked if this animal is indeed the one he secretly selected. If so, the program blatantly crows about its own cleverness. If not, the program admits defeat, and asks the user for a new question with which it augments its data base. The program becomes wiser for the next game.

We will begin by analyzing the overall strategy and data structures used by the program. Then we will go through the code in detail, learning new subrs and techniques as we encounter them.

The program's knowledge of questions and animals (to be distinguished from its knowledge of how to ask questions and how to learn) is expressed by a tree of questions and animal names. The symbol "toptree" is bound to the top of this tree. Animals are represented by the symbol (the unique symbol on the obarray) with the animal's name as its printname. For example, the symbol "buffalo" represents the animal "buffalo" when appearing in the proper context. Questions are represented by lists of symbols, each symbol representing the word which is its printname. For example, the list which prints as (does it have horns) represents that question.

The tree consists of nodes. A node, in this program, is either a symbol representing an animal, or a list of three objects. These objects are, respectively, the representation of a question, a node called the "true branch", and a node called the "false branch". The program operates by starting at the node at the top of the tree, asking the questions, and chasing the "true branch" if the question is answered affirmitavely, or the "false branch" if not. Thus, of the animal(s) on the "false" branch of a particular node the question in that node may be answered "false", and similarly the "true branch". After some game playing, the tree might have a printed representation as follows:

```
((does it have horns)
 ((is it graceful)
  gazelle
  buffalo)
 ((does it growl)
```

```
((is it like a cat)
 lion
 bear)
butterfly))
```

Note how going deeper and deeper into the tree produces animals about which a larger and larger number of statements may be made. The structure of the tree is used to implement a selection process. When the program has progressed down an interaction with the game-player to a given animal, but the player asserts that that animal is not his choice, all of the statements that can be made about the animal in the tree (!) can also be made about the player's chosen animal. Therefore, the addition of a new question and new animal involves only a local change to the data structure, replacing the old animal with the node consisting of the new question, the new animal, and the old animal.

The program has to print out and receive English-language sentences. Dealing with English sentences constitutes a large part of its skill. Sentences are represented by lists of symbols, each symbol representing the word which is its print-name. To make things easier to deal with, all upper-case characters are converted to lower-case characters and user input "re-read" in simulation, this having been done. We will learn how to "re-read" input in this way as we arrive at that point in the program. When sentences are output, the first word is capitalized as is the standard English convention. Often, the program has need to construct sentences from canned sentence fragments (e.g., "(is this animal)") and deduced ones (e.g., "(a bear)"). To facilitate the printing of such sentences, a concept of "constructed sentence" exists. A constructed sentence is either a symbol, representing a single word, or a list of constructed sentences. For example, all of the constructed sentences below are to be printed at the game player as "Is this animal a bear":

```
(is this animal a bear)
(is (this animal)(a bear))
((is this animal) a (bear))
```

and so forth.

Having looked briefly at the two fundamental data structures used by this program, let us consider the source-listing given in this chapter, line by line.

We first note that the source listing appears to be, and is in fact, a print-out of a file in the Multics File System. Lisp programs are usually created by creating files full of forms with a system's regular editors, not by typing forms at the interpreter. This allows for the creation of programs, sets of functions and forms that may be used on different occasions whenever they are needed. We can get the Lisp interpreter to go out to a file in the file system, and

read and evaluate all of the forms in it as though the user had typed them in one by one, expecting their evaluation. There exists a subr called load (1) which is used for this purpose. Applying it to a symbol whose print-name is the representation of a file-name in the operating system under which Lisp is running causes this to happen. For instance, the file containing this program is called "animal". Typing the form

```
(load 'animal)
```

at the interpreter causes Lisp to find that file, and evaluate all the forms in it (not, by the way, printing out the results). This action is called loading the program.

The first character we observe is the semicolon character, ";". It is used for putting comments in Lisp programs. Everything to the right of a semicolon is put there for the benefit of humans reading the program. The Lisp reader ignores all characters to the right of the semicolon. Multiple semicolons may be used stylistically, but only the first one on a given line means anything.

The first comment describes what this program is, and who wrote it and when. This is called journalization, and is useful so that the culprit may be found when the program doesn't operate properly.

The four functions which are next defined, play, explore, grow-in-intelligence, and get-new-predicate, are the heart of the game, and responsible for its characteristic behavior. The remaining functions are support functions, having not to do with animals and trees, but with things like reading sentences, upper-and-lower-case letters, interrogating the user, etc. They are the groundwork upon which the first four functions are built.

The function play is responsible for the highest-level behavior of the program as opposed to the game. It prints out its greeting explaining the game, plays the game once (by calling explore, the next function, which plays the game once), asks if the user wants to play again, and continues doing so until the user answers "no". At this point the user is queried as to whether he wishes to "save" the Lisp world (we will consider that alternative later) and exits Lisp, via the "quit" subr, if he replies in the negative.

The first form in the definition of play applies the princ subr to a symbol with a very long print-name, taking several lines. Symbols with very long print-names are the usual way in Lisp to obtain messages and canned dialogues for user communication. The vertical bar character ("|") which appears to start the symbol name, is not part of

(1) We will stop calling out subrs by name and number, but simply mention them in passing as appropriate.

the name at all, but tells the reader that all characters up until the next vertical bar, are part of the print-name of this symbol. Normally, the first space, dot or newline would terminate the print-name, but this is not so when the vertical bar has been used. For a rather esoteric reason, slashes ("/") must precede all newlines in the print-names of such symbols. The slash does not become part of the printname.

The princ subr is like the print subr, i.e., it prints out on the terminal the printed representation of its argument. There are two major differences between the two subrs:

1) While print outputs a newline before the printed representation of the object and a space after, princ does neither.

2) If there are funny characters in the name of a symbol, such as parentheses, spaces, dots, quote-marks, or newlines, print will slashify them, i.e., put a slash ("/") in front of each such character, while princ will not. Thus, print's output may be read back in by the Lisp reader, for a slash preceding a character removes its special significance. princ's output is intended for human readers, not the Lisp reader, and thus it will not slashify. For instance, if we have a symbol whose print-name has three characters, a close parenthesis, a space, and an open parenthesis, print will give

```
/)/ /(
```

while princ will give

```
) (
```

One usually uses princ to type out messages to a person.

Next we see a new-format do, one with a list of variable-spec's followed by an end-test/exit-form clause. We know this because the first item in the form after do itself is a list, (), which is the same as nil, which is a list of 0 elements. There are thus no variable-spec's. The end-test, the first element of the second item after the do, is nil. When nil becomes non-nil, the loop terminates. As nil never becomes non-nil, the loop does not terminate unless the do is exited by other means. Thus, we have a "do forever". This loop repeats the play-game/ask-for-another-game cycle indefinitely.

The call to explore plays one round of the game. The arguments to explore will become clear when we discuss that function subsequently. The support function ask is used to interrogate the user. It is not a Lisp subr, but a function defined in this file. It is applied to either a single symbol or a constructed sentence. It

asks the user this question, and returns t or nil as he answers yes or no, respectively. ask worries about making sure he only types yes or no, and gives him a hard time for anything else. ask is used as a predicate, albeit a user-defined one. ask with an argument may be viewed as a predicate which goes through the user to get its answer. In this case, when the "user returns nil" for the "wanna try again" predicate, the do is exited, and thus the infinite game-cycle repetition, via the return subr (recall that do is just a special form of prog). Again, we invoke the user as a predicate, via ask, to find out whether to save the game or just quit when he decides not to play any more.

explore is the basis of the game. It is the recursive function that walks the animal-and-question tree. It embodies the algorithm which is the program's game strategy. As it walks down the tree by recursing, it poses the question in each non-terminal node (i.e., node which is a 3-list as described above as opposed to a specific animal) to figure out which branch to recurse over. When it reaches a terminal node, it has arrived at the only animal known to the program consistent with all of the answers that the user gave. It constructs a sentence asking if that is indeed the user's chosen animal, and either performs the blatant crowsmanship, or becomes more intelligent. The becoming-more-intelligent is implemented in another function, grow-in-intelligence.

The two arguments to explore are a node, and that cons of the list which contained this node (i.e., the cons that had this node as its car). This latter argument is provided so that this cons might rplaca'ed with a new node if the program must become smarter. For instance, if explore is visiting the node which prints as

```
((is this animal graceful) gazelle buffalo)
```

it poses the question "Is it graceful?" to the user. If the answer is affirmative, it calls itself recursively with

```
the new node:          gazelle

the containing cons: (gazelle buffalo)
```

the latter being a sublist of the list

```
((is this animal graceful) gazelle buffalo)
```

If the beast is a gazelle, all is fine and good. But if another graceful, horned beast, such as an antelope was chose by the user, then the car of that cons must be changed, by rplaca. Its new printed representation would be

```
(((does this animal rhyme with canteloupe)
  antelope gazelle)
 buffalo)
```

causing the "is this animal graceful" node to look like this:

```
((is this animal graceful)
 ((does this animal rhyme with canteloupe)
  antelope gazelle))
 buffalo)
```

It is for this reason that this cons is passed on recursively.

The first test made by explore is whether it is being invoked upon a specific animal (not a list, i.e., not a cons), or a question-true-false list. The predicate atom is used here for this purpose. In the specific-animal case, the user is queried with a constructed sentence of the form

Is this animal a llama?

A multi-level sentence is constructed by the application of list, resulting in something like

((is this animal)(a llama))

and this is passed on to ask, which uses print-sentence, defined later, to print out such constructed sentences. The support function a-an-hack, applied to a symbol, returns a list of an appropriate article (i.e., "a" or "an") and its input argument. This sentence fragment is combined with the canned "(is this animal)" to make the constructed sentence.

Having correctly guessed the user's chosen animal, explore uses the support function print-sentence to perform the blatant crowing. The two arguments to print-sentence are a (possibly constructed) sentence, and a symbol with a 1-character print-name, being the end-of-sentence punctuation. The punctuation is provided separately because it is different than all of the other words in a sentence. It is not separated by a space from the previous word, and, in fact, is not a word at all, but an artifact of the English implementation of sentences. We note that the question-mark and period symbols are slashified, to remove their special significance to the reader. When there is a special character, like "?", of which we are not sure whether it has special meaning or not, it cannot hurt to slashify it when using it in a symbol name.

If we have posed an animal to the user, and he has said that it is not the correct animal, we must grow in intelligence, by the use of the function of that name. He is passed the animal which it is not, and the cons to rplaca with the new node. We will consider him shortly.

In the case where we are exploreing a non-terminal node, we pose the question in that node, which is the first element of it, the

car of its first cons, and recurse over the selected branch, passing the branch and its containing cons as required.

Note that when explore is called from play, nil is passed as the containing cons. We can do this because we know that the top level of the tree is not a specific animal, but a question-true-false list, by explicit design. Thus, this node can never be a "wrong animal" because it is not an animal at all, and its containing cons need never be rplaca'ed. Thus, we pass an insignificant argument at that point, which will never be used. We must pass something, because explore requires two arguments. The nil is a signal to a human reading the program that this might be some special case, which it is.

grow-in-intelligence is that function which admits defeat, asks the user for the name of a new animal, asks the user for knowledge to differentiate the new animal from the only one known to the program which satisfied all of the posed questions, builds a new node with the question and the two animals, and rplaca's it into the tree in place of the old animal. grow-in-intelligence begins by printing out the "Well, I'm not too sharp.." remark, which asks the user for the name of a new beast. The next bit of logic here is expressed as a lambda-combination, "((lambda (new-beast) ...)" meaning, "With the new beast, let's do these things. We'll answer the question of where the new beast came from later.". It tells a person reading the program that the important stuff of what this function does is right up front, the issue of how it gets this new beast being kind of secondary. The lambda combination is a very good construct for this, for unlike setq's and prog's, the person reading the program does not have to think about where else in the prog the variable "new-beast" may be set or looked at. The lambda expression limits the scope of that variable to the forms inside of it.

The new-beast is represented at this point by a single symbol with a wholly lower-case printname, on the obarray, whose print-name is the name of this new-beast. grow-in-intelligence builds a constructed sentence like this (assume the old beast was a gazelle and the new beast an antelope):

```
((tell me something about)(an antelope)
 (which is not true about)(a gazelle))
```

Again, a-an-hack is used to affix a proper article to the name of the new animal. print-sentence types out this sentence, properly capitalized, with the selected punctuation, a period (".").

grow-in-intelligence calls get-new-predicate to get a list of words from the user which represents the new question, to be built from the user's statement. get-new-predicate must be passed the new animal in order to analyze the sentence fully. We will deal with how he gets this question shortly. grow-in-intelligence then builds the new 3-list via the list subr, applying it to the result of get-new-predicate, the new beast, and the old beast. The new node is

rplaca'ed into the cons where the old animal used to live, in the tree, thereby growing the program's data base.

Now back to the question of where did the name of the new beast come from, which was postponed by the lambda-combination, as being uninteresting. Well, if we look at the line beginning

```
((lambda (animaldesc)
```

we find that the question is again postponed. In this case, "animaldesc" is a list of words representing the animal the user typed in, such as

```
(a bee)
(firefly)
(great horned toad)
```

The task of this lambda expression is to get back a single symbol with the right print-name. For the three examples above, the three symbols will have print-names

```
bee
firefly
great/ horned/ toad
```

Note that in the last case there are blanks in the print-name. I slashify them here only to remind you that this is the print-name of one symbol, not three.

The first simplification that the lambda-expression performs is to remove any article, such as the "a" in "a bee". Since the user's input, at this stage, is guaranteed (by get-statment below) to be a non-null list, asking if its car is one of the symbols "a" or "an" (remember all of the user's input has been interned on the obarray) is reasonable. If so, use the cdr of the user's input, in the case in question being (bee). In all other cases, we use the list as it came. It is the "cond" in the lambda-expression which selects one of these two values for further processing.

The subr explodec creates a list of symbols with print-names 1 character long. If you catenate the print-names of all these symbols together, you get the printed representation of the argument to explodec. It is the human-readable form, such as printed by princ which is used, not the slashified form used by print. For example,

Input to explodec	Output
abcDe	(a b c D e)
(abc def (g))	(/(a b c / d e f / /(g /) /))

The slashes above are so that you can tell the symbol whose print-name is an open paren from an open paren used to represent the start of the list of symbols given above. It is not in the output. The output of explodec given "(abc dec (g))" above is a list of 13 elements (being the number of characters in "(abc def (g))". The symbol whose print-name is an open-parenthesis appears twice in it. The symbol whose print-name is a close parenthesis also appears twice in it. The symbol whose print-name is a space appears twice in it, too. The symbols whose print-names are a, b, c, d, e, f, and g each appear once in it.

explodec is a great way to get at the characters which constitute the printed representation of something. (Of course, that is in terms of the program, not in terms of Lisp, for Lisp does not deal with characters, just symbols (and conses, etc.)). When we apply explodec to a list, as we have seen, the first and last elements of that list are always the symbols with the print-names of an open parenthesis and a close parenthesis respectively. Applying reverse to this list builds a new list of the same single-character symbols in the opposite order. That is to say, if we apply reverse to

(/(b e e /))

we get a list which prints like this:

(/) e e b /( )

which is still a list of 5 elements. Applying the cdr subr to that gets us a list of 4 elements,

(e e b /()

in effect removing the close parenthesis. Applying reverse to that we get

(/(b e e)

putting it forwards. Applying cdr to that, we get

(b e e)

a list of 3 elements, being the characters in the name of the animal. Had we started with "(great horned toad)", we would now have

(g r e a t / h o r n e d / t o a d)

Note the appearance of the symbol with the single-space print-name twice in the above list of 17 elements.

We now want to construct a single symbol whose print-name can be gotten from these lists of characters, from which we have removed the parentheses of the representation of a list, but left spaces.

There is a primitive Lisp function that does precisely this. maknam takes a list of single-character symbols, and constructs a totally new symbol, different from any other symbol, whose printname will be constructed out out the characters given; the symbol will have no binding and an empty property list.

```
(maknam '(r u m b l e))
```

gives a symbol named

```
rumble
```

However, we would like one service which maknam does not perform. We would like the constructed symbol to be interned on the obarray, i.e., if there already is such a symbol on the obarray, (one with this print-name) us it, and if not, put the symbol we construct on the obarray. In this way, any time the user mentions that animal, the symbol will be used to represent it, for the reader interns symbols in this way. The implode does precisely this. It constructs or returns an interned symbol precisely the same way the reader does, but instead of getting the print-name from typed input, we get it from the print-names of the symbols in the list we are given. Thus, we now have either an interned symbol whose print-name is "bee", or one whose print-name is "great horned toad". It is with these symbols that the upper lambda-expression works.

Whew. Now, as to the doubly-postponed question of from where grow-in-intelligence got the original list, i.e.,

```
(a bee)
(firefly)
(great horned toad)
```

which was postponed by the second lambda-combination: it uses a support function, defined later, called "get-non-wisecrack", whose purpose is to get a bunch of words from the user, and return a list of symbols representing this reply. get-non-wisecrack also sorts out remarks which it deduces are not serious. If such a wisecrack is offered to it, it responds with the counter-remark which is the print-name of its argument. We will consider get-non-wisecrack in detail when we get there.

get-new-predicate is the trickiest and most difficult part of the program. It is used by grow-in-knowledge to get a new question to be installed as part of the the tree. Again, it is a lambda combination, which works on the result of get-non-wisecrack, calling the user's statement "wisdom". The user's statement is a sentence true about the new animal, but not the old. (grow-in-knowledge has already prompted the user for this statement). It is the task of get-new-predicate to convert this statement into a question. At this point, the statement is a list of interned symbols, all lower-case. get-new-predicate's result will be a new list of such symbols. The symbols represent words.

Although kind of clever, get-new-predicate is not all that bright. He works best on sentences starting with "it", "it" being the new animal in response to the prompt. Sentences like

it has four legs

become questions like

does this animal have four legs

Sentences like

it eats cauliflower

become questions like

does this animal eat cauliflower

Sentences which state predicate adjectives, like

it is polymorphous

become questions like

is this animal polymorphous

On all other cases, it does not win very well, and constructs a kludgey sentence of the from

would you say that ramafrazz phamblatan

or similar.

get-new-predicate begins by checking if the given sentence starts off with an indefinite reference to a member of the given species, e.g.,

a cow is gentle and loving

The first element of the list representing the user's statement is one

of the symbols "a" or "an", and the second is the new animal (i.e., the interned symbol with that print-name) as passed for this purpose by grow-in-knowledge. If so, the variable "wisdom" is set to a new list, constructed from "it" and the cddr of the old, such as

it is gentle and loving

This allows the rest of get-new-predicate to do better with it.

If the statement now contends that "it has ...." or "it is .....", questions of the form "does it have ..." and "is this animal ..." are formed by appending the portion of the input statement beyond "it has" or "it is" to copies of the pre-canned statement headers just given. The append subr serves admirably here.

If the statement is neither one of having or being, get-new-predicate tries for a sentence stating what the animal does, looking for a verb as the word following "it". It thinks it recognizes a verb by its ending in "s". It is by this means that a statement like

it eats grass

becomes a question like

does this animal eat grass

It is not easy to find a case where this is not so, i.e., a sentence of the form

it xxxxxs ... ...

where xxxxx is not a verb. To perform this analysis, get-new-predicate must analyze the characters in the printed representation of the second symbol in the input statement. For this, our new friend explodec is called into play, and his output reversed. The inside lambda-combination, in get-new-predicate says, "where "revexplode" is bound to the list of single-character symbols in the reversed print-name of the second element of "wisdom", do thus-and-so". So, if we started with

it eats grass

(reverse (explodec (cadr wisdom))), and thus revexplode, evaluates to a list which looks like

(s t a e)

being "eats" spelled backwards. Note that this lambda combination is in the predicate position of a cond-clause; if it returns nil, the cond goes on. Since there are no consequents in this cond clause, if it returns non-nil, what it will return will be the value of the whole

cond, the big lambda combination, and the function get-new-predicate. So first, the lambda-expression starts with an and. If the first form in the and evaluates to nil, not only are the remaining forms not evaluated, but the and evaluates to nil, and thus so does the lambda combination (the inner one), and the cond-predicate is false, and the cond marches on. This first form of the and thus checks to see if it thinks the second word of the statement was a verb, by seeing if the first element of revexplode is the single-character symbol "s". As explodec interns the single-character symbols it returns, we can ask for this "s" in the program, the interned "s", knowing we will be asking about the one explodec would be expected to return.

So the and sees if the sentence is of the verbal kind it thinks it knows about. If the and turns up nil, the cond falls through. If, however, the second word of the response appears to be a verb, get-new-predicate tries to reconstruct the verb by stripping off the "s" before building the question. The progn, being the second clause of the and, is alwas evaluated in its entirety if the first clause of the and is non-nil, whether or not any forms in it evaluate to nil. It is progn, not and. Immediately, the "s" is chopped off by

```
(setq revexplode (cdr revexplode))
```

Thus,

```
(s t a e)
```

becomes

```
(t a e)
```

Now a check is made for some common verbs that end in a doubled letter, an "e", and then an "s" in the third-person singular, such as "buzzes". The three-clause and inside the progn checks for two conditions, an "e" being the last letter (now the first element of revexplode, the "s" having already been removed), and the next two elements being the same letter. If these two conditions are true, the final clause of the inside and is evaluated, and it strips off the "e". Thus,

```
buzzes
```

became

```
(s e z z u b)
```

then

```
(e z z u b)
```

and finally

```
(z z u b)
```

The new verb is now constructed by implodeing the reversed "revexplode" as it now stands, stripped of "s" and perhaps "e". Thus, we get the symbol

buzz

A new statement fragment is constructed out of this and the rest of the original statement, giving

give milk

if the original statement were

it gives milk

This is appended to a copy of the canned statemnt-header "(does this animal)", resulting in the final question, which can never be nil, and thus, the progn, the and and the lambda combination all return a non-nil result.

We have now described the heart of the program. We must now describe the guts. All of what is left are the support routines, used by the four (!) functions we have just described.

As we go on through the listing, we next encounter a form which is not a function definition at all. However, like function definitions (and all else) appearing in a file, it is evaluated at the time the file is loaded. This particular form gives all of the symbols whose print-names are single letters of the alphabet properties that allow the program to differentiate between upper and lower case letters, and translate between them where needed. The names of these properties are "upper" and "lower". Each symbol whose print-name is an upper-case letter gets the corresponding symbol with the same letter lower-case as a print-name as its "lower" property. Similarly, the latter symbol gets the former as an "upper" property. No other symbols in the Lisp world of this program will have "upper" or "lower" properties. For example the interned symbol "Z" has a "lower" property of the interned symbol "z", and the interned symbol "q" has an "upper" property of the interned symbol "Q". We emphasize the fact that these single-character-printname symbols are interned; it is this which allows us to identify them in the code of the program (i.e., ask, "is it the interned "s"?").

These properties are given by the nameless function specified by the lambda expression in this form. It gives its first argument a "lower" property of its second, and its second an "upper" property of its first. We do this for all of the letters in the alphabet by means of the very powerful and useful subr mapc, which was explained earlier. We give mapc three objects in this case, a function (which can be either a lambda expression or a symbol which has a subr or lambda expression properly attached to it) and as many lists as that function expects arguments (in this case, two). What

mapc will do is march down the two lists in parallel, taking one element from each of them at a time, and apply the function to these to objects (the elements chosen from each list). Thus,

```
(mapc '(lambda (x y)(print (cons x y)))
      '(A B C)
      '(a b c))
```

would print

```
(A . a) (B . b) (C . c)
```

Here, we apply this double-property-putting function to the lists

(A B C D E F G H I J K L M N O P Q R S T U V W X Y Z)

and

(a b c d e f g h i j k l m n o p q r s t u v w x y z)

which were gotten by applying our friend explodec to symbols whose print-names were the upper and lower case alphabets. Instead of printing out a cons, we cross-relate the properties.

Next in the file is a function which makes use of these properties, to ensure that a word is all lower-case letters. Applied to a symbol representing a word, presumably gotten from user input, it returns a symbol (perhaps the same one) whose print-name contains only lower-case letters. That is to say, "foo", "FoO", and "FOO" all cause "foo" to be returned. Its basic technique, like other functions we have seen so far, is to blow apart its input with explodec, do something with it character-by-character, and squeeze together a (possibly new) symbol with implode. Here we use mapc's brother, mapcar, which is even more powerful. mapcar again takes a function, and as many lists as that function expects arguments (in this case, one, as the function is (lambda (y)..)) and applies the function to list elements in succession. The difference between mapc and mapcar, remember, is that instead of returning something useless, mapcar returns a new list, whose elements, in succession, are the successive results of the successive function applications that mapcar brought about. (1)

In get-lower-case, the function we "map" over the list of single-character-print-name-symbols is "If you are upper case, we want your lower-case. Otherwise, you'll do.". In the context of our

(1) One is tempted to ask, "Why use mapc at all if mapcar is so much better?". The answer lies in the fact that construction of the new list is relatively expensive, and is to be avoided if you are not going to use it. Using mapc also tells people reading the program that you do not intend to use the result.

program, this function is written in Lisp as

```
(lambda (y) (or (get y 'lower) y))
```

This says "Where y is the letter under consideration: If its "lower" property is non-nil (then it must be upper-case), the answer is that (i.e., its "lower" property, which is its lower-case translation). Otherwise, use the letter as it stands.".

implodeing the result of mapping this function over the characters in the symbol produces a suitably lower-cased symbol which (as we pointed out of all implode results) is interned.

The function capitalize sort of does the opposite. Given a symbol representing a word, it returns a (perhaps other) symbol whose print-name begins with a capital letter. Thus, "foo" gives "Foo", but "FOO" gives "FOO". Only the first letter is dealt with. This is done again by explodecing the symbol, checking the first element of the result for an "upper" property, replacing this element with its "upper" property if it has one, and implodeing the result. If the first element didn't have an "upper" property, we just use the input symbol as it was given, to avoid the work of implodeing when we know the answer.

The function print-sentence implements a very powerful primitive, and is sort of hairy. It takes a (possibly constructed) sentence and a punctuation mark (being a single-character symbol) as arguments. The sentence is printed out, with spaces between words, the first word capitalized, and the punctuation mark printed afterwards. print-sentence has a couple of features that are not even used in this program. When one writes such a function, one should consider all special cases, such as being passed a sentence of 0 elements, i.e., nil, or nil as a punctuation mark. In these cases, print-sentence prints nothing for the sentence or punctuation mark, respectively.

print-sentence, like many hairy recursive functions, is broken down into two parts, an outer-level non-recursive function which handles the special cases and an inner "gut", a recursive function which is called by the outer function with appropriate arguments. In this case, the outer function, print-sentence worries about the cases of nil sentence or punctuation, the printing of the punctuation, the issuing of a newline (the subr terpri does this) and the calling of the recursive part, guts-of-print-sentence. print-sentence also special-cases being passed a single symbol other than nil as an argument; if this is the case, it is simply princed out with the punctuation, without being capitalized, as it is assumed to be some kind of verbatim message, i.e., not a constructed sentence.

guts-of-print-sentence is where the real work is done. He is applied to lists, never to symbols. print-sentence applies it to his input when he knows that not to be a symbol, and

guts-of-print-sentence calls himself recursively when he finds that an element of one of these lists is a list.

guts-of-print-sentence mapc's his internal lambda expression over his input list. This causes each symbol in it to be princed out, and each list in it to be recursed over. The only really subtle thing going on is the flag "firstflag", which is passed to any invocation of guts-of-print-sentence as t if and only if it is known that the first real word, i.e., not sublist, but real word, has not yet been printed out. This flag triggers the captalizing of the first word to be printed when it is t, and causes each word to be printed to be preceded by a space, when it is nil. Note that print-sentence passes it to guts- (for short) as t, because in this case it certainly is known that nothing has been printed out. Each invocation of guts- turns off its idea of "firstflag" after it has processed its first element, be it symbol or sub-list. For after this first element was processed, the first word had to have been printed, whether this invocation or a successive one actually printed it. The local idea of "firstflag" is handed down as a starting-point for all recursive invocations. Once nil, it can never become t. If this confuses you, you must convince yourself by staring at these two functions, and trying sample executions either at your terminal or in your head. It is subtle.

get-statement is straightforward. His task is to read a line of user input from the console, and return it as a list of lower-cased words. He also insures that the user does not type an empty line, pestering him if he does. get-statement is implemented as a do, with one local variable, "statement", initially bound to nil. The loop terminates, returning the value of "statement", when the latter becomes non-nil. (1) The form beginning with "(setq statement" is that which does the real work, reading a line of characters as a single object with readline. readline reads an entire line of characters, up to and including a newline character, from the user's console. It returns a symbol whose print-name consists of this character string. (2) We apply explodec to this result to get a list of characters, including all of the spaces and the newline, single-character-print-name-symbols like all else. We build a new list out of a pair of parentheses, with all of the characters from this explodecion in the middle. This now looks just like the printed representation, or should we say, a possible printed representation, of the list we are trying to build. The subr readlist takes such a

(1) We say "(not (null statement))", although we could have said just "statement" in this end-test as well. We feel that the form given here may be clearer, and for sure, in compiled code, it is as efficient. We will talk about the compiler later. "null" is the same as "not".

(2) On Multics, a "string object", i.e., not a symbol, is actually returned, but the effect of the subsequent explodec is identical. We will talk about "string objects" later.

list of character-symbols, makes a character string out of them, and asks the reader to read it! This is sort of like the inverse of explodec but for arbitrary lists, not just symbols (like implode is). The result returned from readlist is the list of symbols representing words that the user typed in. This is a very useful technique to learn, because we have used the Lisp reader to solve all problems of decomposing and parsing the user's input, but we did not require him to type parentheses around a list.

get-statement then maps the function get-lower-case, already described, over all the elements of the reconstructed list read in, and proceeds to deal with the guaranteed-lower-case-symbol list. If the user typed only a blank line, or a newline alone, we would have given readlist a set of parentheses with only a newline or spaces and or a newline between them. When we type such characters to the reader, we get nil, which is what readlist as well will return for a pair of parentheses separated by whitespace. If this is the case, the value of "statement" will be nil at the or. The or checks for this, and needles the user via print-sentence. The do will go around again until a non-empty line is typed.

The function ask asks a question, typing it out via print-sentence, supplying the "?" punctuation, and getting a list of typed-in words via get-statement. Its do terminates when a line starting with the word "yes" or "no" is typed. It translates "yes" and "no" into t and nil, so that it may be used as a predicate. There are no new Lisp concepts in this function.

get-non-wisecrack is another do that reads statements until something it likes, in this case non-nil is produced. A non-empty statement is read via get-statement. A nameless function, represented by the lambda-expression in get-non-wisecrack, is mapped over each word in the gotten sentence. The entire mapc is encased in a prog, whose value becomes the value of the variable "statement". If the internal lambda-expression finds any word it dislikes, it prints out the caller-supplied rebuff, and causes the entire prog to return nil at once, causing the do to repeat. If all of the words in the sentence pass this censor, (return response) is evaluated, causing the prog to return the guaranteed non-nil response. The variable "response" is set to this value, and the do returns this object to get-non-wisecrack's caller.

a-an-hack creates a list of an indefinite article and its input argument. Its internal logic ought be quite clear at this point.

The next form in the file sets the value of the variable "toptree" at the time the program is loaded, to the initial data-tree, It is a 3-list of one question and two animals. This question will always be the first question the program asks. Note that the variable "toptree" (whose value, you may recall, is passed by play to explore) is not a lambda or prog variable of any function. It is called a free or special variable.

The function save-game is called to put the current Lisp world in a huge burlap bag called a "saved environment" and put that bag in a file called "animal.sv.lisp" in the current directory of the Multics File System. This burlap bag can be opened up and made into a Lisp world identical to the one at the time it was "saved". This is done by issuing the command

lisp <filename>

to Multics, where <filename> is whatever argument "save" (which is an fsubr, and thus does not have its argument-forms evaluated for it by eval) was given.

And when the bag is opened, Lisp does this thing: it mapc's eval over the binding of the symbol "errlist", i.e., evaluates all the forms in this list of forms. Thus, save-game sets this variable to a list of the form "(play)", which causes the function play to be invoked when the user invokes lisp in this manner. Thus, the user of the animal game does not have to know how to cause the top-level function play to be invoked; Lisp does it for him. In fact, he does not have to know anything about Lisp at all to play the game.

When this program is first loaded, play must be invoked by hand, i.e., by giving the form "(play)" to the interpreter. When we do this, the game will give its little litany about how to play it, and, on Multics, respond immediately with "Eh? Whazzat you say?" even though we didn't type anything. This is because the first call to readline produces the newline which was typed by the user after the form "(play)". This newline looks, to the program, like an empty line. We had to type this newline, or else Multics would not have sent the line "(play)" to Lisp. This is not a problem when the game is started up from the saved environment because no newline is typed to Lisp in this case.

The last form in the file is somewhat arcane. It too, is evaluated at load time. It changes the internal tables used by the reader to make comma and period be treated as whitespace. This is done so that arbitrary punctuation thrown in by the user is ignored, instead of becoming part of symbol names or indicating conses (as period normally does in printed representations). The programmable reader is a useful feature of Maclisp. We will not go into it here. Read up on it in the manual to learn more.

Creating and Debugging LISP Programs in the Multics Environment

For all but the simplest exploratory toying with LISP, sitting around and typing forms at the interpreter is not a reasonable way to input programs. The interpreter is very unforgiving about typing the wrong thing, and once you type the right thing, only the paper and the property lists know what you have typed. Hence, it is usual to prepare a file containing LISP forms (function definitions and other forms) using a traditional editor (edm or qedx on Multics) and cause it to be read into LISP. For example, one might prepare a file by saying:

```
!edm myfuns
 Segment myfuns not found.
 Input.
!(defun foo (x) (cons (x (gensym))))
!.
 Edit.
!w
!q
 r 233 0.245 35 162
```

(! marks a line that is typed by the user)

To load this into LISP, one applies the "load" SUBR to the pathname of the file:

```
!lisp
 *
!(load 'myfuns)
 t
```

to get lisp to read and evaluate the forms in this file. Once this has been done, we may hand the interpreter forms containing applications for the functions defined therein. There are two common errors one will encounter in this procedure, having too many parentheses or not having enough. If there are not enough closing parentheses at some point in a file, usually some object (a list) will not finish before the file runs out. Hence the message

```
lisp: End of file in middle of object
```

may be taken as a hint that this is the case. On the other hand, too many closing parens at some point often cause atoms in a suceeding form to appear at top level, i.e. to be read and evaluated by load, as one of the forms in the file. In this case,

```
lisp: undefined atomic symbol: cons
```

or something similar is usually the result. When any of these errors

happen, or the error from the file's name being misspelt or not found, lisp "stacks" the error (in a very shallow stack) and expects you to take some corrective action. This usually is hitting the "BREAK" or "QUIT" key and typing a "g" in response to "CTRL/". This cryptic formula causes lisp to unwind the error stacked up, and start again reading, evaluating and printing at top level. Again, see the manual for more details.

When you run your functions and find that they don't work, you would probably like to look at the definitions you have provided. Looking at the values of symbols is not a challenge, since simply typing the name of a symbol at the interpreter causes it to be evaluated (its value retrieved), and the value printed.

There exists a function (an fsubr) called grindef (grind definition, so called because of the amount of work it must do), is available to take a function definition, and print it out as a "defun" form with the conventional Lisp indentation.

```
Thus (grindef foobar)          ;the cadr of the form is the
                               ;function to be ground
```

might print out

```
(defun foobar (x y) (prog (a b c)
                          (cond ((eq a 'what)
                                 (setq b 7)
                                 (go cc))
                                (t (cond ((atom sp) .......
```

The value returned by "grindef" is a very peculiar symbol whose printname mysteriously doesn't print at all.

Now once you find your error, you might want to fix your program. You can quit LISP (apply "quit" to no arguments), and take care of it, but it is usually most convenient to invoke an editor from lisp and re-apply load to read the new file. To do this, we use the subr cline, which may be applied to a symbol whose print-name is the Multics command you want to execute. For instance, one can say

```
(cline '|edm myfuns|)
```

to cause the editor to edit the source file, without leaving lisp. When you exit the editor, cline returns nil and you can then reload the source file.

If looking at your program, its behavior and its variable values is not sufficient to find your problem, you might want to trace your functions. The FSUBRs trace and untrace exist for this purpose.

If you say

```
(trace fun1)
```

the tracing package will print out a message each time fun1 is applied to anything, and will also print out the objects to which it was applied. Furthermore the tracing package will print out the value retraced by fun1. As you might have guessed, untrace turns tracing off. For further information about the tracing package which, in fact, is quite versatile, check out the manual.

```
;;;
;;;  Animal game program, by Bernard Greenberg 1/88/78.
;;;

;;;
;;;       Basic top-level function.
;;;

(defun play ()
       (princ '|Let's play a game. Choose a random animal.|) (terpri)
       (princ '|I'm gonna try to guess it by asking you questions,|) (terpri)
       (princ '|and you give me yes-or-no-answers. OK? let's go.|) (terpri)
       (do () (nil)                                              ;do forever.
           (explore toptree nil)
           (or (ask '|That was fun. Wanna try again|)
               (return nil)))                                    ;If he is done, escape from the "do".
       (and (ask '(do you want to save me))                      ;If he wants to save it, do that.
            (save-game))
       (quit))

;;;
;;;       The gut of the whole game. That which recurses over each node.
;;;

(defun explore (node what-to-rplaca)
    (cond ((atom node)                                           ;We are at a specific animal
           (cond ((ask (list '(is it) (a-an-hack node)))
                  (print-sentence '(hey hey I sure am clever/, huh) '/?)
                  (print-sentence '(lisp MUST be a great language) '/.))
                 (t                                              ;Time to learn some new knowledge.
                  (grow-in-intelligence node what-to-rplaca))))
          (t                                                     ;Not an animal, but a question.
           (cond ((ask (car node))                               ;pose the question
                  (explore (cadr node) (cdr node)))              ;Explore the true branch.
                 (t (explore (caddr node) (cddr node)))))))      ;else, do the false branch.

;;;
;;;       Artificial intelligence implemented here.
;;;

(defun grow-in-intelligence (loser howd-we-get-here)             ;learn
    (princ '|Well, I'm not too sharp today. I give up.|) (terpri)
    (princ '|Just what kind of beast did you have in mind?|) (terpri)
    ((lambda (new-beast)
             (print-sentence (list '(tell me something about)
                                   (a-an-hack new-beast)
                                   '(which is not true about)
                                   (a-an-hack loser))
                             '/.)
             (rplaca howd-we-get-here                            ;change the old node.
                     (list (get-new-predicate new-beast)         ;build new node
                           new-beast
                           loser)))
     ((lambda (animaldesc)
              (implode (cdr (reverse (cdr (reverse               ;strips off ()
                                           (explodec
                                            (cond ((memq (car animaldesc) '(a an))
                                                   (cdr animaldesc))
                                                  (t animaldesc))))))))
      (get-non-wisecrack '|Hey, thats not the name of a real beast.|))))
```

```
;;;  Get a new predicate by munging the English of the statement.

(defun get-new-predicate (new-animal)                          ;get what to decide on.
    ((lambda (wisdom)
             (cond ((and (memq (car wisdom) '(a an))
                         (eq (cadr wisdom) new-animal))
                    (setq wisdom (cons 'it (cddr wisdom))))) ;a cow is.. -> it is
             (cond ((eq (car wisdom) 'it)                      ;Best case.
                    (cond ((eq (cadr wisdom) 'has)             ;Winninger yet.
                           (append '(does it have) (cddr wisdom)))
                          ((eq (cadr wisdom) 'is)
                           (append '(is this animal) (cddr wisdom)))
                          (((lambda (revexplode)
                                    (and (eq (car revexplode) 's)
                                         (progn (setq revexplode (cdr revexplode))
                                                (and (eq (car revexplode) 'e)
                                                     (eq (cadr revexplode)(caddr revexplode)) ;"buzzes" -> "buzz".
                                                     (setq revexplode (cdr revexplode)))
                                                (setq wisdom (cons (implode (reverse revexplode))
                                                                   (cddr wisdom)))
                                                (append '(does this animal) wisdom))))
                            (reverse (explode (cadr wisdom)))))
                          (t (append '(is it so that) wisdom))))
                   (t (append '(would you say that) wisdom))))
     (get-non-wisecrack '|Aw, be serious. I asked you a real question.|)))

;;;
;;; Functions for hacking case-ness, i.e., upper/lower of words.
;;;

;;; Executed at load time.  This function gives each lower case symbol
;;; an upper case symbol as its "upper" property, and vice versa.

(mapcar '(lambda (x y)
                 (putprop x y 'lower)
                 (putprop y x 'upper))
        (explodec 'ABCDEFGHIJKLMNOPQRSTUVWXYZ)
        (explodec 'abcdefghijklmnopqrstuvwxyz))

;;;  Get lower case from possibly partially-upper-case word.

(defun get-lower-case (x)                                      ;x is the symbol for the word.
    (implode
     (mapcar '(lambda (y)                                      ;Get y's "lower" property, if any, otherwise y.
                      (or (get y 'lower) y))
             (explode x))))

  ;; capitalize.

(defun capitalize (w)
    ((lambda (exploded)                                        ;The exploded word.
             ((lambda (first-letter-upper)                     ;"upper" property of the first letter.
                      (cond (first-letter-upper                ;If there is one, use it
                             (implode (cons first-letter-upper (cdr exploded))))
                            (t w)))                          ;If not, just return w.
              (get (car exploded) 'upper)))
     (explodec w)))
```

```
;;;
;;;  Function to print out a sentence. All lists are linearized.
;;;

(defun print-sentence (sentence punctuation)
    (cond ((null sentence))                                    ;do nothing if nil at this level.
          ((atom sentence)                                     ;not a list.
           (princ (capitalize sentence)))                      ;single word
          (t (guts-of-print-sentence sentence t)))             ;recurse, hard case.
    (and punctuation (princ punctuation))                      ;Print if non-nil
    (terpri))                                                  ;newline

(defun guts-of-print-sentence (sentence firstflag)
    (mapc '(lambda (x)                                         ;For each element of the sentence,
                   (cond ((atom x)                             ;a word, print it.
                          (or firstflag (princ '| |))
                          (princ (or (and firstflag (capitalize x)) x)))
                         (t (guts-of-print-sentence x firstflag)))       ;a list
                   (setq firstflag nil))                       ;It is not the first time anymore.
          sentence))


;;;
;;;       Functions to read in a line
;;;

(defun get-statement ()
    (do ((statement))
        ((not (null statement)) statement)
      (setq statement
            (mapcar 'get-lower-case
                    (readlist (append '(|(|)
                                      (explodec (readline))
                                      '(|)|) ))))
      (cond ((null statement)
             (princ '|Eh? Whazzat you say?|)
             (terpri)))))

(defun ask (query)              ;get yes or no answer
   (print-sentence query '/?)
   (do ((response (car (get-statement)) (car (get-statement))))
       ((memq response '(yes no))
        (eq response 'yes))
     (print-sentence '(|Hey, can you give me a yes or no answer|) '/?)))

;;;
;;;       Toys and games.
;;;

(defun get-non-wisecrack (remark)
    (do ((response))
        (response response)
      (setq response (get-statement))
      (setq response
            (prog ()
                  (mapc '(lambda (x)
                                 (cond ((memq x '(i you hell damn go the if dont))
                                        (print-sentence remark nil)
                                        (return nil))))          ;Exit prog with nil.
                        response)                                ;What to map over
                  (return response)))))                          ;If mapc didn't return, get ans.
```

```
(defun a-an-hack (word)
    ((lambda (first-letter)
             (list (cond ((memq first-letter '(a e i o u))
                          'an)
                         (t 'a))
                   word))
     (car (explodec word))))
;;;
;;;  Initialize the game.
;;;

(setq toptree '((does it have horns) buffalo butterfly))

;;;
;;;       Save the game if wanted.
;;;

(defun save-game ()
    (print-sentence '|Type:   lisp animal| nil)
    (print-sentence '|to play this game again| '/.)
    (setq errlist '((play)))                          ;makes game self-starting
    (save animal))

;;;
;;;       Cause period and comma to be ignored.
;;;

((lambda (syntax)
         (sstatus syntax 56 syntax)                   ;56 is an ascii "."
         (sstatus syntax 54 syntax))                  ;54 is an ascii ","
 (status syntax 40))                                  ;40 is an ascii " "
```

Notes on the Programming Language LISP

by Bernard Greenberg

Part IV

Notes on the Programming Language LISP

PART 4

This portion of the notes concerns itself with several diverse topics which make LISP a more interesting programming language.

&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&

Lexprs

We have met many subrs which can be applied to a variable number of arguments, such as "+" and "list". Most subrs, such as "cons", are applied to a fixed number of arguments (in this case, two). We have learned how to define functions; yet, so far, they can take only a fixed number of arguments. For example, the function fg15 below takes 3:

```
(defun fg15 (abel zflag dontcpush).....
```

We would like to be able to define functions that take a variable number of arguments. Such functions are called lexprs, for they are like the list subr, which takes many arguments. (In fact, such subrs are called lsubrs in this context.) Functions of a fixed number of arguments are called exprs in this context.

We define a lexpr by specifying a single (non-nil) symbol instead of the argument list in the function definition instead of the usual lambda list. For example,

```
(defun mylexpr n
            (cond ((...
```

As would be expected, a lambda expression like

```
(lambda n (cond ((..
```

gets filed under the expr property of the symbol mylexpr.

When such a lambda-expression is applied, the symbol "n" is bound to a fixnum being the number of arguments to which the lexprish lambda expression was applied. A lexpr always wants to know to how many arguments it was applied. This, however, does not solve the problem of actually finding out to what arguments it has been applied. A lexpr may obtain its arguments by means of the arg subr. arg is applied to a fixnum, being the number of the argument that you want. For example, if we have

```
(defun blorph n
          (print (arg 2)))
```

and we evaluate the form

```
(blorph 'boy 'is 'this 'random 'stuff)
```

we will find that "is" is printed out. For another example,

```
(defun fumble n
          (do i n (1- i) (= i 0)
              (print (arg i))))
```

will print out "stuff", "random", "this", "is", and "boy" on successive lines.

More List Processing Functions

Often it is useful to see if two pieces of list structure are shaped the same and look the same. For instance, a list might be as follows:

```
((the king of spain)
 (the thing of main)
 (the ring of bane))
```

representing a bunch of things we had determined useful for some purpose. Now each of these things has "substructure" in some sense that we care about. For example, suppose "ainlist" were bound to the list above. Then

```
          (cadddr (car ainlist))       gives "spain"
          (cadddr (cadr ainlist))      gives "main"
   and    (cadddr (caddr ainlist))     gives "bane".
```

Now suppose that we are accumulating a list like this of random four-word, four-syllable posessives that end in something rhyming with "ain". Suppose that a part of this system had come to the conclusion that "the zing of rain" could, or should, also be an appropriate member of this list. It has come to this conclusion by considering various properties of "rain", "zing" and maybe something that a user had typed in. So it has developed the list

```
(the zing of rain)
```

by building it up. We would like to see if this fact is already in "ainlist". Well, you may recall that "memq" could be used for such things. However, (say "x" is bound to "(the zing of rain)")

```
(memq x ainlist)
```

will not do it. In fact, just because "(the king of spain)" looks like a member of the list "((the king of spain)" (the thing of main..... " does not mean it is. The problem is that "(the king of spain)" is a bunch of characters printed on paper, not a piece of list structure. These characters do not identify any single piece of list structure in any LISP world. Furthermore, in any given LISP world, there may be any number of different pieces of list structure that print out "(the king of spain)". In fact, any cons whose car is the symbol named "the" and whose cdr is a cons whose car is the symbol named "king" and whose cdr is a cons whose car is any symbol named "of" and whose cdr is a cons whose car is a symbol named "spain" and whose cdr is nil, will print out just like that!

Perhaps we don't care if they are the same physical piece of list structure or not, but just if they look the same or not! Well, there is a predicate that tells you this, given that the symbols used in each are physically the same symbols. This predicate, called equal,

is applied to two pieces of list structure. Basically, if they look the same when printed, equal returns "t", otherwise "nil". Thus if "x" is bound to "(the king of spain)" and "y" is bound to the "(king of spain)" then

```
(equal x y)
```

returns "t", given that the "the" used is both expressions is the same symbol "the", and that "king" is the same symbol named "king", etc. This function "equal" is most valuable in pattern matching applications and in the general class of problems where assertions or facts which are equivalent may be independently derived or constructed by differing parts of a program.

equal acts exactly as if it had been defined in Lisp as follows:

```
(defun equal (x y)
    (cond ((eq x y) t)
          ((and (fixp x) (fixp y) (= x y)) t)
          ((or (atom x) (atom y)) nil)
          (t (and (equal (car x) (car y))
                  (equal (cdr x) (cdr y))))))
```

The function member is just like memq, but uses "equal" instead of "eq" as a basis to determine whether or not the first argument is a member of the second argument. Hence,

```
(member x ainlist)
```

will correctly determine if "x", bound to "(the zing of rain)" is a member of ainlist, as above or not.

"delete" is like "delq", but uses equal as a comparison.

* *

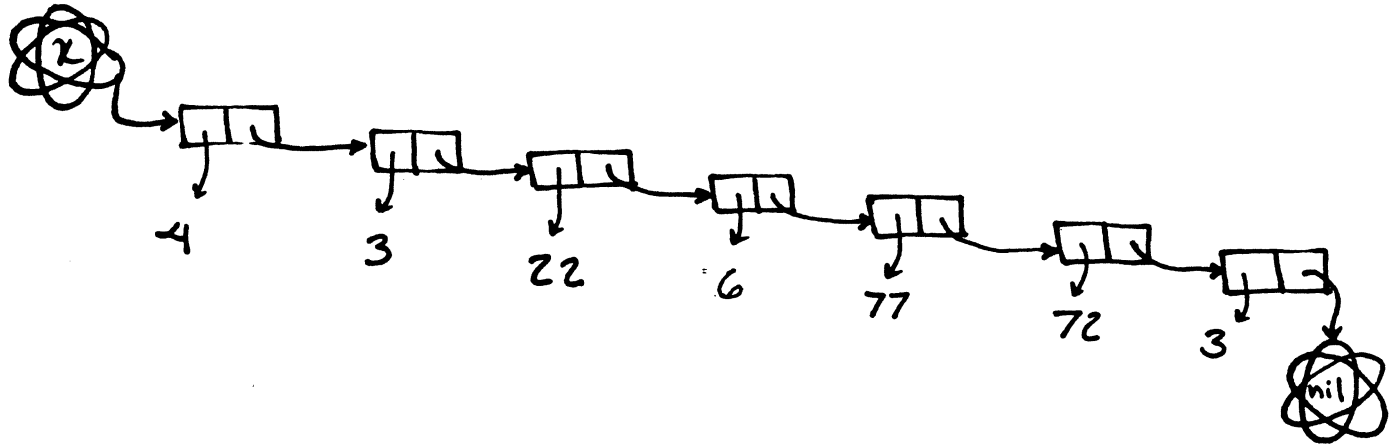
Another very useful class of functions is the sorting functions, which sort lists or arrays (see the later discussion of arrays) based upon arbitrary criteria. The criterion is expressed as a function (like in mapc or mapcar), and the sorting function applies this function to determine ordering of the list. As in mapc, these may be name-symbols or lambda expressions. The list is sorted by patching various conses around, until it has the cons at its head whose car is the lowest-sorting element. The cdr of this cons is the next lowest-sorting element, etc.

For instance, suppose we have

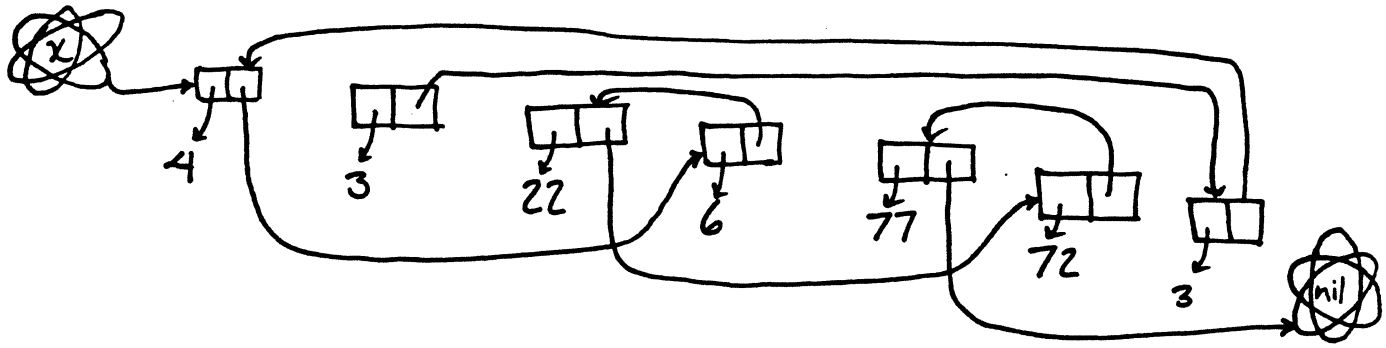
```
(setq x '(4 3 22 6 77 72 3))
(sort x '<)
returns          (3 3 4 6 22 72 77)
```

Note that "x", which was bound to the cons whose car was 4, still is. sort sorts a list by applying the given predicate to the elements of the list, to sort the list into order.

Before:



After:



It is a very common error to supply a form like

```
(sort x '>)
```

expecting x to wind up bound to something sorted. As can be seen in the above illustration, it will not. In all such cases, a form such as

```
(setq x (sort x '>))
```

is what you will have meant. sort is applied for its returned value

as well as its side-effects.

sortcar sorts the things in a list by applying the predicate supplied to the cars of the elements of the list. For instance,

```
(sortcar '((z Zebedee Zachariah Zeke)
           (a Andy Able Art)
           (b Baker Bill))
         'alphalessp)
```

returns

```
((a Andy Able Art)
 (b Baker Bill)
 (z Zebedee Zachariah Zeke)).
```

(It is necessary to explain that the predicate alphalessp compares the printnames of its two symbolic arguments and returns "t" if the first collates lower than the second, else "nil".) This function is usually used to sort lists of things, where the thing at the head of the list determines its sorting position.

< < < < < < < > > > > > > >

More Exotic Object Types

Maclisp provides a wide variety of object types. So far, we have only considered four: symbols, conses, fixnums, and subrs. Symbols have printnames, bindings, and property lists; conses have cars and cdrs; fixnums have magnitude; and subrs can perform actions.

As well as providing fixnums to represent integers, Maclisp provides flonums (for "floating point numbers") to represent real numbers. They are like fixnums in that they have magnitude, and nothing else. The magnitude of a flonum is a real number. Like fixnums, they evaluate to themselves, and their printed representations are the intuitive representations of real numbers. For example, 60.5 2.67, -65.2, 35e6 are typical ones. The base of representation of flonums is always ten: they are decimal.

There are also bignums, or infinite precision fixed numbers. Any number which looks like a fixnum, but is actually bigger than a machine word on the computer, is a bignum. The ability of LISP to handle bignums like 32367263076123737362736702137112773672630772673616336636 3 is incredible and allows arbitrary precision computation of mathematical constants, etc. The reason that there exists a distinction between fixnums and bignums is that fixnums are much easier for the machine to deal with. If the generality of bignums is not needed, it is advantageous to deal only with integers representable in a machine's native integer format. Like fixnums, the printed representation defaults to octal, with a point indicating decimal.

The subrs for numbers that we have learned about +, -, *, / are limited to fixnums. They will not work on flonums, bignums, or numbers of differing type. There is a similar set of subrs for flonums, named +$, -$, *$ and /$. There is no set for bignums, but the general arithmetic subrs, named plus, difference, times, and quotient work for any or intermixed types. (= works for fixnums or flonums but not mixed types. This peculiarity is due to the PDP10 floating point number format.)

Although the general arithmetic subrs will work for all kinds of numbers, it is more efficient to use the specific ones (+ or +$) as they are faster and the compiler (see the later discussion of the compiler) can produce better code. Of course, you can only use the specific subrs if you know you will be dealing only with fixnums or flonums but not both in some form.

Lisp also provides predicates called floatp and numberp. floatp returns t if its argument is a flonum, and numberp returns t if its argument is any kind of "number" (a fixnum, flonum, or bignum). Also, you should know that fixp returns t if given a bignum; most of the time, you aren't interested in distinguishing between fixnums and bignums. If you are, there exists bigp which returns t if its

argument is a bignum.

(MULTICS ONLY)

Another basic type of object is the character string object. These are objects that have nothing but printnames. Like numeric objects, they self-evaluate. Their printed representation is their printname, surrounded by double-quotes. Any type of character at all may appear in the print-name without being slashified; if a double-quote character appears in the print-name, it is doubled in the printed representation (that is, printed out twice).

"dkdkjbs ksdkj uuu iiisud(((((""

is a good string object. One can say

(setq x "dkdkjbs ksdkj uuu iiisud(((((")

as opposed to

(setq x '"dkdkjbs ksdkj uuu iiisud(((((")

because character string objects are self-evaluating. Character string objects are usually used for printing messages: princ prints them by typing out the print-name character for character for what it is worth: the double-quotes are not printed, and quotes inside the string are only printed once. In the animal program in Part III, we used the vertical-bar character ("|") to obtain symbols with peculiar printnames for the purpose of printing messages. Although the vertical-bar works in all Maclisp implementations, character string objects are usually used for such purposes on Multics.

There are functions to deal with the character string objects, to make little ones from big ones, one out of several or several out of one. For instance,

(substr "ablebaker" 2 4) gives "bleb"

just like the PL/I builtin.

(catenate "abcde" "fghijkl" " " "xxx") gives

"abcdefghijkl xxx"

All the character string functions take either character strings or symbols as input, in the latter case using the printname. They always return character string objects. Hence

(catenate 'foo) yields "foo"

Note that the Multics command line function, cline, only likes to be applied to character string objects.

(index "foobarshabaz" "arsh") gives 5,

the index (position of the second argument is the first, like the PL/I builtin.

(NOT MULTICS ONLY)

A very useful object is the array object. An array object is a very funny kind of object. As opposed to having a binding, or a car and a cdr, all of which are ways of designating other objects, it has a numerically indexed array of designations. That is, it has its first, second, third,... forty-seventh (which are all objects) as opposed to its car or cdr. Now, to get the car of a cons, we apply the car subr to the cons. To get the "33rd" of an array, we apply the array itself to 33. An array is a type of function: it is a basic functional object, like a subr or a lambda expression. An array can be applied to a fixnum, and it will hand back an appropriate object. Array objects are hung off the array property on some symbol.

Say we had a 40-cell array off of the symbol "Harray". Suppose further that the 32nd of "Harray" was "(a b c (u i))". Then

(Harray 32) gives (a b c (u i))

eval knows about arrays, as they are functional properties. It evaluates the forms in the cdr of the form mentioning the array, as it does for a subr. It knows how to apply them to fixnums. (The array object is actually a little subroutine that knows how to keep its own house!).

As we have rplaca and rplacd to change the car and cdr of conses, and set and setq to change the bindings of symbols, we have the fsubr store to change array cells....

(store (Harray (+ 6 3)) 'Hoohah)

makes the symbol "Hoohah" be the 9th of the array property of "Harray". store is an fsubr because it uses the array reference "(Harray (+ 6 3))" to figure out where to store something, as opposed to evaluating it and doing something with the resulting object.

Arrays are created by putting an array object on a symbol, which can then be used as a function. The fsubr array does this.

(array foo t 30) creates an array object dimensioned 0 to 27 off of foo. This single dimensioned array can be applied to one fixnum.

(array bar t 32 45) creates a 2 dimensioned array from 0 to 31 and 0 to 44. The "t" is much too hairy to explain now: consider it necessary. Note that array is an Fsubr as it deals with bar, not bar's value.

The Callable Evaluator

The functions eval and apply, in terms of which we have been describing the evaluation process, are actually subrs which are callable from any Lisp program. eval takes one argument and apply two.

eval, as we learned about long ago gets applied to something you want to evaluate, just as "cdr" gets applied to something whose cdr you want. Thus, if "x" is bound to "(+ 3 6)" then

```
(eval x)
```

causes "(+ 3 6)" to be evaluated giving 11 (octal).

```
(eval 'a)
```

gives the binding of the symbol "a". One can express the basic loop of the LISP interpreter at top levels as

```
(do () (nil)    ;do forever
  (print (eval (read))))
```

If you are using eval in an elementary LISP program you are probably doing something wrong. Note that we did not need it to do anything until now. You only need eval if you are writing a program which does something with Lisp, as opposed to blue eyes, Fibonacci numbers, or symbolic expressions. For instance, suppose you had an interactive subsystem which processed commands of some sort, and was written in Lisp. As a convenience you might want to allow the user to type in limited Lisp forms to perform simple calculations, or disturb the environment in some way. In this case, after having read in something that you determined you wanted to be interpreteted as a Lisp form, you might want to apply eval to it to cause it to be interpreted.

eval is the basis of the interpreter. The Lisp interpreter is essentially a loop handing read-in forms to eval and printing out the results. eval is the basis of the definition of Lisp.

:::::::::::: :::::::::::: ::::::::::::

Apply is the right hand of eval. eval and apply call each other back and forth to perform Lisp interpretation. We have learned very thoroughly what it means to apply a subr, lambda expression or array to a set of objects. Forms are a way of specifying how to find what objects a function is to be applied to, with eval being the agent responsible for getting these objects by interpreting the form and getting apply to apply the required array/subr/lambda expression to the gotten objects (This is but one more concise statement of the entire evaluator).

If you have a bunch of objects that you obtained by whatever means, that you wish to apply some function to, you may apply apply to the function (function-propertied symbol or lambda expression) and a list of the things to which you want the function applied. For instance, suppose you had an interactive subsystem which read "commands" such as

```
          fire Charlie
          make-boss Max John
  and     print-depends Irving
```

to manipulate some kind of data base. The data base might maintain all kinds of employee data. Via a technique such as that used in get-statement in the animal program of Chapter III, we can easily convert these into lists, i.e.

```
          (fire Charlie)
          (make-boss Max John)
  and     (print-depends Irving)
```

Suppose we had functions make-boss, fire and print-depends, such that

```
          (fire (get current-employee 'manager))
```

at some point in the program for this system would fire the employee who is the manager of the current binding of "current-employee". Well, if the user typed in to the system

```
          make-boss Max John
```

(say "x" were bound to the list constructed from this, i.e. "(make-boss Max John)"), we could not say "(eval x)" to cause "Max" to be made the boss of "John", because if we evaluated

```
          (make-boss Max John)
```

Lisp would attempt to apply make-boss to the current bindings of the symbols "Max" and "John", which is not what we want. We want to apply make-boss to the symbols "Max" and "John" themselves. So, we apply apply to

```
          make-boss
 and

          (Max John)
```

by saying (that is, by evaluating)

```
          (apply (car x) (cdr x))
```

since (car x) is make-boss and (cdr x) is "(Max John)", the list of

objects to which "make-boss" is to get applied.

apply is applied to two arguments. The first may be be a name-symbol of any function, or a lambda-expression. The second is a list of objects to which that function is to be applied.

```
(apply 'cons '(a b))
```

returns

```
(a . b)
```

just as does

```
(cons 'a 'b)
```

The Lisp Compiler

Up until now we have been talking about the Lisp interpreter, a subsystem which reads forms, and evaluates them via a well defined procedure. As we know, this consists of looking at forms to find objects to which functions should be applied, evaluating the forms within them telling what objects should be gotten and applying them to functions. Applying subrs, involves executing machine language programs. Applying lambda expressions involves saving symbol bindings, setting new bindings, and evaluating forms in the bodies of lambda expressions.

Although the original development of LISP stemmed from the Interpreter, as we have become familiar with it, the interpretation of Lisp programs is not the most efficient way to carry out the computations and manipulations they express. Remember that a Lisp program is nothing but a collection of specifications (forms) of how to get objects to which certain functions should be applied.

It is possible to write a program (almost always a Lisp program) which analyzes a LISP program and produces a machine-language program to manipulate Lisp objects in the way the program specifies. Such a program is called a Lisp Compiler. For example, the function

```
(defun sumsquare (x y)
    (+ (* x x) (* y y)))
```

can be translated into a machine-language subr which, applied to two fixnums, returns a fixnum being the sum of the squares of the input arguments. In general, the compiler reads a file full of "defun" forms and creates a machine-language program containing subrs that do the same things as the Lisp forms in the original file did. Since the compiler figures out how to get what to apply to what, and how to keep track of intermediate results, the interpreter need never be called during the execution of such a subr (of course, if the subr calls eval explicitly, that doesn't count).

Note that the previous paragraph said that the little function "sumsquare" will be translated into a subr which sums squares. In doing so this subr will simply pick up its arguments, square one, save the result, square the other, add it, and return the result. It will not locate the symbols x and y, save their previous values, or for that matter disturb or use them in any way. In fact, if sumsquare called some other function which looked at the symbols "x" or "y", it would not find them bound to the arguments of sumsquare. "x" and "y" are thus true variables in the compiled subr, as opposed to true bound-symbols of a lambda list. Note what an elegant implementation of "where x is the first quantity and y the second" we have here: symbols named x and y are never involved at all.

It is via the compiler that the user creates his own subrs.

If we want, we can make the compiler actually go through the work of making "sumsquare" save the old values of "x" and "y" and rebinding them to the arguments of sumsquare. If we do this, other functions called will find that "x" and "y" are indeed bound to what they should be. In this case, the variables so specified are called special variables (although all variables in interpreted forms act this way). Normally, lambda variables of functions need not be special, unless they are used for communicating betweeen functions (the variables, not their values. That is to say, if one function expects to find a value in a variable that another function set). Special variables are less efficient than the other kind, local variables.

One can define Lisp perfectly consistently as a language translated by a compiler into machine code, used for manipulating Lisp objects, and never need or mention the evaluator and its artifacts, eval and apply. Most "production" Lisp programs are compiled, and are often as efficent as compiled programs in "traditional" languages.

The compiler reads forms from a file. It considers each form in turn: if the form appears to be a definition of a function (a form whose car is defun), the compiler analyzes it and generates a subr of the same name. If the form is a list whose car is the symbol "declare", THE COMPILER ITSELF EVALUATES, AT THE TIME IT SEES THIS, ALL THE FORMS IN THE CDR OF THIS LIST. This is possible because the compiler is written in LISP and can apply eval, or any other function it chooses to whatever it wants, as can any LISP program. Thus, if the form

```
(declare (print 'hello-im-compiling-foo))
```

appears standing in your file, the compiler will print out

```
hello-im-compiling-foo
```

at the time it sees it. This is generally used in a more useful way to tell the compiler things, like what variables are special. There is a function in the compiler called special, which is an fsubr, which is used for just this. The cdr of the form invoking special is a list of the symbols that you would like to have declared special. For instance,

```
(declare (special x y y1 q) (special v))
```

declares all these variables as special. In the animal program in Chapter III, a decalaration like

```
(declare (special toptree))
```

would be appropriate.

The fsubr declare does nothing. Hence, if you read a file

with a declare in it into LISP, the declare will be ignored. The compiler, however, is not applying declare to anything. When it sees a list whose car is declare, it does this special thing that we have just discussed.

All forms which are not functions are simply copied in encoded form into the object program. These are known as "random forms" in Lisp compiler parlance. They will get evaluated when the object program is loaded into Lisp.

The compiler itself is a very large and beautiful Lisp program and is one of the more interesting things that have been done with Lisp. On Multics, the compiler is invoked by saying

```
lcp Nicholas
```

where Nicholas.lisp is the name of a file containing function definitions to be translated. The resulting object segment is called Nicholas, for example.

On ITS, we compile programs by invoking NCOMPL, the ITS Lisp compiler by saying

```
:NCOMPL
```

NCOMPL responds by saying

```
LISP COMPILER 703  BY 702  IN OLDIO
_
```

to which we respond, after the "_", with the FNAME1 and FNAME2 of the file to be compiled, and the magic incantation "(fk)"... i.e.,

```
_NICHO LAS (FK)
```

The Macro Facility

One of the most potent and fascinating features of Maclisp is the macro facility. This facility constitutes a form of language extensibility. One can construct one's own language out of Lisp, and have it compiled by the compiler or interpreted by the interpreter.

The motivation for macros comes out of the compiler. One can clearly implement an interpreter for any language whose constructs are expressible as list structure in Lisp (implementing Lisp in Lisp is a well-known example). However, convincing the compiler to compile such things is not obviously easy.

Rather than for the total construction of new languages, the macro facility is usually used to extend Lisp, adding new constructs built out of old ones.

"do" is a good example of a builtin macro. We showed in an earlier chapter an expansion of a "do", a set of forms containing portions of the original do form.

Suppose we had a program that often used forms like

```
(setq alist (cons thrung alist))
or  (setq mv12 (cons (+ 32 yy mvn) mv12))
or  (setq wt17 (cons (caddr (ay by)) wt17))
```

or in general,

```
(setq ***something*** (cons ***something-else*** ***something***))
```

This is a very common construction in Lisp programs, as it is the operation of pushing something onto a list. We should like to be able to say

```
(push this-thing that-list)
```

when we mean

```
(setq that-list (cons this-thing that-list))
```

However, the chances of writing the function "push" are nil, so to speak. For as soon as we have said

```
(defun push (thing list)...
```

we have lost. For although we will pass the list and the item to be pushed to the function "push", there is no way that this function can ever perform the setq of the symbol that-list in the form that called "push". Macros provide a way to achieve this functionality, and more.

A macro is a function which is called upon to translate the

form in which it appears into some other form. The result of the macro-function is a form, which is used in place of the original (macro) form by either the interpreter or the compiler as appropriate. Macros are defined just like regular functions, as follows:

```
(defun push macro (x)
     (list 'setq (caddr x)(list 'cons (cadr x)(caddr x))))
```

Note that a macro always gets one argument, which is the form in which it appeared.

Evaluating this defun gives "push" a macro property of "(lambda (x)(list 'setq..." etc. Now the interpreter, upon seeing a form whose car is a symbol with a macro property, says:

"Jeez, I don't know what this thing even means. However, my programmer has given me a lambda expression which will translate it for me into something I understand. So I'll apply that lambda expression to this form, and work on what comes back instead!"

So, eval calls his friend apply, and

```
(lambda (x)(list 'setq (caddr x)
                       (list 'cons (cadr x)
                                   (caddr x))))
```

is applied to

```
(push      (+ 32 yy mvn) mv12)
```

which, to this lambda expression is just another piece of data. If you yourself apply that lambda expression to that list, you, as eval, will get

```
(setq mv12 (cons (+ 32 yy mvn) mv12))
```

which is exactly what you want. Note that eval reconsiders the answer returned by a macro as a form in place of the original. The macro did nothing with the values of yy, mvn, or mv12, and performed no additions. It only messed around with a form IT understood, to TRANSLATE it for eval.

An answer returned by a macro can have other macros in it, or maybe even references to itself. As long as eval, reducing it (applying macros each time it gets one until it's not a macro form anymore) ultimately gets something that's not a macro form.

The first incredible thing about macros is that the COMPILER is willing to invoke your macros at the time he is compiling your programs, so you can tell him what your stuff means. That is to say, the compiler, which is a Lisp program, will invoke your code during compilation to help him in his task. Every macro can be thought of as a little piece of a Lisp compiler. Hence, if you define and use

"push" in your program, it will be as efficient as had you put the setq and cons in there instead.

The second incredible thing about macros is that the entire power of the language is available to them. To help your macros organize the meaning of their forms, you can use any function in Lisp, builtin or of your own construction. You can call other-language programs, or cause your programs to be compiled differently at different times of day. The power of the macro facility derives from the fact that Lisp code is Lisp data, and as such is trivially easy to deal with. Think of the complexities of handling PL/I code (ASCII or EBCDIC characters strings) in PL/I, or FORTRAN code in FORTRAN.

The third and perhaps most incredible thing about macros is the ability to define abstract and complex languages that bear little relation to Lisp, except in parentheses. Since Lisp is capable of expressing just about anything computational, one can write forms using macros, whose translation into Lisp can be just about anything computational. One can tailor the language to define constructs that suit any given application, and have it be compiled into code as efficient as the result of explicit coding in the basic primitives of Lisp.

"Functions compute, macros translate."
-D. A. Moon

A Close Parenthesis

It is hoped that these notes have provided a taste of the true flavor of Lisp. Rather than concentrate on developing competency in Lisp, we have chosen to expose the interested student to the basic concepts, and a few usable programs, such that he or she might at least say, "Well, Lisp, that's a bunch of stuff pointing to other stuff, and it's really good for making models of things, or simulation."

We have tried to show you how Lisp car-and-cdr worlds are a more reasonable representation of the things that make life interesting than fixed decimal (15) or FILE OLDMSTR RECORD IS PAYROLL. It is hoped that you can at least extrapolate in your mind what kind of neat things one can do with this. Our sincerest hope is that you will see a piece of that part of the computer programming world where the computer has become a tool whereby man extends his mind and his own grasp of it.

LISP - A Radical Introduction (Greenberg notes)

Worksheet #1

The class notes and lectures for this course will go a long way towards introducing you to the Lisp language, and with the aid of these worksheets it is hoped that you will get some "hands-on experience" with Lisp as well. The worksheets are designed to reinforce the material covered in lecture, supplement some of the more picky details not spelled out in the notes. and raise additional questions in the student's mind to lead him to more advanced examples.

Public terminals are available in many dorms, the student center library, and at delphi in building 38 (room 344). To use Multics from a 300 baud-rate terminal turn on the terminal (halfduplex), dial extension 8-8313, press the "DATA" button on the modem, replace the receiver back on the dataset and press "linefeed" on the terminal. Multics will type an introductory message such as:

```
Multics 33.0: MIT, Cambridge, Mass.
Load = 17.0 out of 85.0 units: users = 17
```

You may then login with your Personid and Projectid FOLLOWED BY A CARRIAGE-RETURN AND A LINE-FEED as follows:

```
login JDoe SIPBIAP
```

Substituting YOUR OWN PERSONID for John Doe's in the example above. Multics will then ask for your password, which you must again follow with a carriage-returns and a linefeed. After you have logged in successfully, typing the word "lisp" (all lowercase) will start up the Lisp interpreter.

You are now in a read-eval-print loop. Lisp will read in any form you type, evaluate it, and print the value it returns.

Below are a series of Lisp forms to evaluate. You may choose to try and work them yourself and then check you answer by using the computer, or possibly you will want to have the computer evaluate the forms in order to help you understand a new concept that you are having problems with. Experiment! Try your own problems; this is one of the best ways to get use to using and understanding Lisp.

NOTES:

To correct typing errors, use "number-sign" (#) to delete the last character typed and "at-sign" (@) to delete the current line. When you first login you must follow each line with a carriage-return AND linefeed. If you are using a decwriter you may issue the command "la36" before starting up Lisp, after which only a carriage-return OR a linefeed is required. This may also be accomplished after Lisp has been invoked by having Lisp evaluate: (cline "la36") .

To get out of an endless loop or to get back to Lisp if things don't seem to be going right simply hit the "break" key once (sometimes labeled "quit" or "attn") and when Lisp types "CTRL/" then you type the letter "g" (followed by a carriage-return and linefeed) and you will be back in Lisp's read-eval-print loop.

The character slash "/" is used by Multics Maclisp to quote certain characters (for instance, to enter a symbol whose name has a parenthesis or dot or space in it). Therefore in order to use the "/" subr to divide you must enter two slashes. For example: to divide 6 by 2 you would enter: (// 6 2).

You may get out of Lisp and back into Multics by typing "(quit)" and you can get out of Multics by typing "logout" and then hanging up the telephone.

Your account on Multics may be used from 6:00 PM until 11:00 AM on weekdays and all day Saturday and Sunday.

For more detailed information on how to use Multics, a free set of "Notes on Using Multics" is available from the Student Information Processing Board.

If you should run out of funds, or would like to do a project or just learn more about Lisp after the course is over, apply for money at the Student Information Processing Board.

For help with any questions or problems call the Student Information Processing Board at extension 3-7788 or come in to the office in room 39-200.

```
(set (quote a) 6)
a
(setq b 5)
b
(+ 4 5)
(quote (+ 4 5))
(+ a b)
'(+ a b)
(* 3 4)
(* (+ a 3) (+ b 4 5 6))

a
(quote a)
'a
(symeval (quote a))
(symeval 'a)
(eval 'a)

(set  (quote colors) (quote (red green blue)))
colors
(car colors)
(cdr colors)
(cons (quote red) 'yellow))
(setq firstcolor (car colors))
(setq paints (cons (quote yellow) colors))

()
nil
(car nil)
(cdr nil)
(setq smallcons (cons (quote foo) nil))
(car smallcons)
(cdr smallcons)


(quote a)
'a
'(a . b)
'(a . nil)
'(a . ((b . nil ) . nil))
'((a . (b . nil)) . c)

(setq alph6 '(a b c d e f ))
alph6
(car alph6)
(quote alph6)
(cdr alph6)
(cddr alph6)
(cdddr alph6)
(cddddr alph6)
(cadr alph6)
(caddr alph6)
(cadddr alph6)
```

```
(setq pair (cons 'left 'right))
pair
(car pair)
(rplaca pair 'wrong)
(car pair)
(rplacd pair 'correct)
(cdr pair)
(rplaca pair alph6)
(rplacd pair nil)
(rplacd pair 'alph6)


(// 6 2)
(setq eqn '(* (+ a b) (- 4 (* 3 a (// 4 b)))))
(car eqn)
(cdr eqn)
(quote eqn)
(eval 'eqn)
(eval eqn)
(cadr eqn)
(caadr eqn)
(caddr eqn)

(setq d 'e)
(setq e (quote f))
(set (quote f) 4)
(quote d)
d
(symeval d)
(eval d)
(eval (quote d))
(eval e)
(eval f)
(eval (eval d))
```

Lisp - A Radical Introduction

Worksheet #2

NOTES:

What to do when an error occurs.

If Lisp finds that it is about to add two conses together, or evaluate a symbol that has no binding, or any other illegal action that should cause an error to be printed, Lisp places you back into a read-eval-print loop (as usual) WITHOUT unbinding anything. This allows you to examine the bindings of various symbols as they were bound when the error occured. In order to undo these bindings you can send a control-g to Lisp. On Multics this is done by hitting the "break" key once (sometimes labeled "quit" or "attn") and when Lisp types "CTRL/" then you type the letter g (followed by a carriage-return and linefeed) and all of the TEMPORARY bindings that were set at the time the error occured will be forgotten. This was not terribly important on worksheet 1 when no temporary bindings were made. Beginning with worksheet 2 you will be defining and using your own functions. It is important to release temporary bindings when an error occurs. To better understand what happens try the following example:

```
(defun    zort (x y z)
          (progn    (print x)
                    (print y)
                    (print z)
                    (print w)))
(setq x 15)
x
(zort 10 20 30)
x
y
z
w
CTRL/g    ; hit "break" then "g" then "carriage-return linefeed"
x
```

Changing the base used for reading and printing fixnums:

The symbol "base" is bound to a fixnum which determines what base numbers are printed in. Similiarly the symbol "ibase" is bound to a fixnum which the reader uses to determine what base numbers read into lisp are in. Therefore evaluating the forms (setq base 10.) and (setq ibase 10.) will cause lisp to read and print fixnums in decimal rather than octal.

```
; PROBLEM 1
; Evaluate the following forms:

(> 5 4)
(> 4 6)
(< 3 7)
(< 4 4)
(= 5 (+ 3 2))
(setq a 6)
(setq b 4)
(> (+ a b ) (- (* a b ) a))

(not (> 6 3))
(null (> 6 3))
(not (not (> 3 6)))
(null (not (> 3 6)))
(null nil)
(null (quote nil))
(null 7)

(setq smb 'foo)
(set (quote lst) '(foo bar baz))
(set 'fxn 17)
(symbolp smb)
(symbolp lst)
(symbolp fxn)
(symbolp (quote lst))
(symbolp (quote (a)))
(atom smb)
(atom fxn)
(atom lst)

(or t nil)
(or (> 5 6) (< 4 3))
(or nil 5)
(or (atom lst) 'lst-is-not-an-atom)
(or (atom smb) 'smb-is-not-an-atom)
(setq num 3)
(cond     ((= num 2) 'two)
          ((= num 3) 'three)
          ((= num 4) 'four))
(cond     ((atom lst) lst)
          (t 'lst-is-not-an-atom))
(and t nil)
(and (> 5 4) (< 3 4))
(and (< num 4) (> num 2) 'num-is-3)

(setq alph (list  'a 'b 'c 'd 'e))
(setq bet '(f g h i j k))
(setq fred '(sam))
(cons fred bet)
(cons 'fred bet)
```

```
(append fred alph)
(append alph bet)
(list alph bet)
(cons alph bet)
(append alph (list alph bet) nil bet '((end)))

(memq 'a alph)
(not (null (memq 'a alph)))
(memq 'a bet)
(memq (car fred) fred)
(delq 'h bet)
bet
(append alph (list (car bet) (cadr bet)) (cons 'h (cddr bet)))

'(lambda (x) (+ 1 x))
((lambda (x) (+ 1 x)) 4)
(defun incr (x) (+ 1 x))
(incr 6)
(setq incr 'december)
(grindef incr)
(grindef (quote incr))
incr

; PROBLEM 2
; Use defun to create a function that switches the order of 2
; elements in a list the same way that the following lambda
; expression does it.

((lambda (x) (list (cadr x) (car x)))
 '(a b))

; PROBLEM 3
; Presented below are three functions for computing factorial:
; fact0, fact1, and fact2.  Fact0 is a recursive function taken
; directly from the definition of factorial.  The second, fact1,
; is an iterative fortranesque example using prog.  The final
; example is also an iterative implementation using the new style
; "do" function.  Note that there is no body to the do.  Write
; three separate functions similiar in style to the three
; factorial functions that compute the sum of a list of fixnums.
; Thus (sum '(4 6 3 5 )) should evaluate to 22 (octal) and
; (sum '()) should evaluate to zero.  "sum0" should be recursive,
; "sum1" should be iterative and use prog, and "sum2" should use
; the do function.
```

```
(defun fact0 (n)
       (cond ((= n 0) 1)
             (t (* n (fact0 (1- n))))))

(defun fact1 (n)
       (prog (result)
             (setq result 1)
        label(cond ((= n 0) (return result)))
             (setq result (* n result))
             (setq n (1- n))
             (go label)))

(defun fact2 (n)
       (do ((i n (1- i))
            (result 1 (* result i)))
           ((= i 0) result)))


; PROBLEM 3
; The Lisp function "reverse" will create a new list with the top
; level elements reversed.  Thus (reverse '(a b (c d) e)) returns
; (e (c d) b a).  Write your own version of reverse called rev1.
; [Harder problem]: Also write a function rev2 which reverses
; elements at all levels of a list.  Thus
; (rev2 '(a (b c (d e) f g) (h i) k)) should return:
; (k (i h) (g f (e d) c b) a).


; PROBLEM 4
; The function "delq" is a destructive function -- that is rplaca
; and rplacd are used to actually delete the specified element
; from the given list.  Below is a function "remove1" which
; returns a new list with the requested element removed.  Write
; an iterative function which uses "do" that performs the same
; task.


(defun remove1 (thing a-list)
       (cond ((null a-list) nil)
             ((eq thing (car a-list))
              (remove1 thing (cdr a-list)))
             (t (cons (car a-list)
                      (remove1 thing (cdr a-list))))))
```

Lisp -+ A Radical Introduction

Solutions to Worksheet #2

NOTES:

Among the things you should have noticed from evaluating the forms given in the first problem are the following:

a) The functions "null" and "not" are identical.

b) The functions "and" and "or" do not always evaluate all of their arguments. In other words, they are faubrs like "cond", "do", and "prog". Although they are useful for performing logical operations, they are often used to control orders of evaluation.

```
(and (= x 4) (go somewhere))
```

is like saying

```
(cond ((= x 4) (go somewhere)))
```

but is simpler. Thus, evaluating (or 3 barf) will not cause an error even if "barf" is unbound.

c) We apologize for giving you the example (grindef (quote incr)) in the first problem. Those of you who sat there waiting for "grindef" to return something soon learned that "grindef" returns a symbol whose name does not contain any characters at all. (You, too, can get at this mysterious beast simply by doing (implode nil), for whatever that's worth.) The function "grindef", as we have now learned, does the right thing when given symbols which are defined as functions or bound to other objects.

d) When Lisp reads in a fixnum with a trailing period, the number is assumed to be in base 10. This is why evaluating (setq base 10.) will convert the output base to base 10 or (setq base 16.) to hexadecimal. Evaluating (setq ibase 10) will never change the value of "ibase", regardless of what its current value is. (Think about it.)

PROBLEM 2

The idea here was to use defun to create a new function which would perform the same action as the given lambda expression when placed in the functional position of a list to be evaluated. In other words, create a function "switch" such that (switch '(a b)) --> (b a).

```
(defun switch (list) (list (cadr list) (car list)))
```

PROBLEM 3

A couple of points are worth noting here: The recursive definition of "sum" is not only the shortest, but is also the one which most clearly represents the algorithm which we are trying to implement. We also learned earlier that the subr "+" can take a variable number of arguments. A call to this subr takes place by means of evaluating a list whose car is the symbol "+" and whose cdr is a list of the arguments we wish to pass to this function. We can easily write a function which will create and evaluate such a list:

```
(defun sum (list) (eval (cons '+ list)))
```

Recursive definition:

```
(defun sum0 (list)
       (cond ((null list) 0)
             (t (+ (car list) (sum0 (cdr list))))))
```

Prog definition:

```
(defun sum1 (list)
       (prog (result)
             (setq result 0)
        loop (cond ((null list) (return result)))
             (setq result (+ result (car list)))
             (setq list (cdr list))
             (go loop)))
```

Iterative definition:

```
(defun sum2 (list)
       (do ((l list (cdr l))
            (result 0 (+ result (car l))))
           ((null l) result)))
```

PROBLEM 4

It was possible to write these functions in several different ways, but we have given only the recursive and iterative definitions below. Again, we notice that the recursive definitions are far more straightforward than their iterative counterparts. Note also that the iterative definition of "rev2" does contain a recursive call -- it would be quite hairy trying to write "rev2" without any recursion whatsoever. Finally, note that the definitions for "rev2" test only to see if their arguments are atoms, not null lists as the definitions for "rev1" do. This is because "nil" is an atom as well as a list, and (cond ((atom x) x)) is identical to (cond ((null x) x)) if "x" is nil.

```
(defun rev1 (list)
       (cond ((null list) nil)
             (t (append (rev1 (cdr list)) (list (car list))))))

(defun rev1 (list)
       (do ((old-list list (cdr old-list))
            (new-list nil (cons (car old-list) new-list)))
           ((null old-list) new-list)))
```

```
(defun rev2 (list)
      (cond ((atom list) list)
            (t (append (rev2 (cdr list)) (list (rev2 (car list)))))))

(defun rev2 (list)
      (cond ((atom list) list)
            (t (do ((old-list list (cdr old-list))
                    (new-list nil (cons (rev2 (car old-list)) new-list)))
                   ((null old-list) new-list)))))
```

PROBLEM 5

Several things are worth noting here:

a) In all of the examples we have given here, the form of "do" which is used is the "new-style" type, which, as you recall, uses the following syntax:

```
(do ((⟨first var⟩ ⟨initial value⟩ ⟨repeat value⟩) ... (⟨nth var⟩ ...))
    (⟨end test⟩ ⟨exit form⟩ ...)
    ⟨body⟩)
```

where the body of the "do" is a sequence of forms to be evaluated during each step of the iteration. However, none of our examples which use "do" contain a body -- they require nothing more than manipulation of index variables.

b) A useful technique which does not appear in other languages is that of giving the variable of iteration a binding which is not necessarily a number. In the cases we've given, our variables of iteration have been bound to lists.

c) Note also that the expressions that appear as ⟨*initial value*⟩ and ⟨*repeat value*⟩ can be arbitrarily hairy. Expressions which are commonly used as repeat values are (1+ i) or (cdr i), but anything can be used. Note, for example, the repeat value of new-list in the following function, i.e. (append new-list ...).

```
(defun remove (thing a-list)
      (do ((old-list a-list (cdr old-list))
           (new-list nil (append new-list
                                 (cond ((eq thing (car old-list)) nil)
                                       (t (list (car old-list))))))
          ((null old-list) new-list)))
```

Lisp -- A Radical Introduction

Worksheet #3

PROBLEM 1

Evaluate the following forms:

```
(list 'a 'b 'c)
'(a b c)
(list 1 '(a . b) 'fred 'one)
(list)
(list 1 nil 3 t)
(list (list (list nil)))

(append '(how are you) (list 'today '?))
(append (list 1 2 3) nil '(4 5 nil 6))

(putprop 'fred 13 'age)
(get 'fred 'age)
(get 'fred 'owes-back-taxes)
(putprop 'fred 20000 'owes-back-taxes)
(get 'fred 'owes-back-taxes)
(plist 'fred)
(remprop 'fred 'owes-back-taxes)
(get 'fred 'owes-back-taxes)

(get 'car 'subr)
(defun frobnicate (x) x)
(get 'frobnicate 'expr)
(get 'frobnicate 'subr)

(print '(a b /.c))
(princ '(a b /.c))
(print '|Section 5.2|)
(princ '|Section 5.2|)

(explodec 'fred)
(explodec 'a/.b)
(explodec '|I am a funny symbol.|)
(explodec '(1 2 3))
(implode '(a b c))
(implode '(h i | | t h e r e))
(implode (explodec '|Hello, how are you?|))

(mapc 'princ '(m a s s a c h u s e t t s))
(mapc 'princ (explodec 'massachusetts))
(mapcar '1+ '(2 4 6 8))
(mapcar 'fixp '(a 3 x barf (a b) 4))
(mapcar '(lambda (x) (+ 47 x)) '(2 4 6 8))
```

```
(mapcar '+ '(2 4 6 8) '(-2 -4 -6 -8))
(mapcar 'cons '(one two three) '(1 2 3))
(mapcar '(lambda (x y) (putprop x y 'father))
        '(USA PoBach Isaac)
        '(George JSBach Abraham))
(get 'USA 'father)
```

The following two functions are being introduced here and have not been mentioned previously. The function "^" takes two fixnums and returns the exponentiation of the first to the second. Note that since the magnitude of a fixnum must be an integer, all results returned will be rounded to integers. The function "length" takes one argument, a list, and returns the number of elements in that list. Here are some forms you can type in to see what these guys do.

```
(^ 5 2)
(^ 4 -1)
(length '(1 g a 47 bernie))
(length '(1 (2 3) 4))
(length '())
```

PROBLEM 2

Use defun to create a function "concatenate" which will take two symbols as arguments and will return a symbol whose name is the concatenation of the names of the arguments. In other words, `(concatenate 'hello 'there)` --> `hellothere`.

PROBLEM 3

On the last worksheet you defined a function "rev2" which reversed a list and all of the elements of the list which were themselves lists, i.e. `(rev2 '(a (b c) d e))` --> `(e d (c b) a)`. Lisp has a built in function (a subr) called "reverse" which returns a list with only the toplevel elements of the list reversed, i.e. `(reverse '(a (b c) d e))` --> `(e d (b c) a)`. Using "mapcar" and "reverse", rewrite "rev2" -- your new definition should be much simpler than your earlier one.

PROBLEM 4

Write a function which will sort a list of numbers. Hint: Define <u>two</u> functions -- one which sorts a list in a recursve manner, and another which takes a number and a sorted list of numbers and returns a new list with the number inserted in the appropriate place. It is easiest if both functions are recursive. Note also that if we substitute the predicate "alphalessp" for the predicate "<", we can easily modify the function to sort symbols alphabetically.

PROBLEM 5

Below is a function which makes extensive use of property lists and mapping functions. Try to guess what it does.

```
(defun who (person)
       (do (((list-of-lists (mapcar '(lambda (x) (get x 'kids)) (get person 'kids))
                            (cdr list-of-lists))
             (new-list nil (append (car list-of-lists) new-list)))
            ((null list-of-lists) new-list))))
```

Now set up a small data base to work with -- the function which is the first argument to "mapc" will give each element of the second argument a "kids" property which is the corresponding element of the third argument.

```
(mapc '(lambda (father sons) (putprop father sons 'kids))
      '(abraham isaac ishmael jacob esau)
      '((isaac ishmael)
        (jacob esau)
        (kedar abdeel nebaioth)
        (reuben simeon dan levi napthali issachar judah gad asher zebulun joseph benjamin)
        (eliphaz reuel jeush jalam korah)))
```

If you can't figure out what's going on, here are a few examples to try --

```
(get 'abraham 'kids)
(get (car (get 'abraham 'kids)) 'kids)
(who 'abraham)
(who 'isaac)
```

PROBLEM 6

Write a symbolic differentiator which can handle expressions containing addition, multiplication, and exponentiation. The first argument to this function should be the expression which is to be differentiated, and the second argument should be the variable to which the differentiation should be performed with respect to. Arithmetic expressions are most easily represented as lists, with the car of the list being the operator, and the cdr being the operands (as usual). For example, 3x+4 should be represented as (+ (* 3 x) 4). The basic rules you should implement are listed below:

$$\frac{dx}{dx} = 1$$

$$\frac{dy}{dx} = 0 \quad y \neq x$$

$$\frac{d}{dx}(u+v) = \frac{du}{dx} + \frac{dv}{dx}$$

$$\frac{d}{dx}(u*v) = v\frac{du}{dx} + u\frac{dv}{dx}$$

$$\frac{d}{dx} u^v = v u^{v-1} \frac{du}{dx} + u^v \log u \frac{dv}{dx}$$

An important thing to note here is that your differentiator should not actually attempt to perform the addition of two expressions, simply create the list structure which represents their addition. For example, assuming your function were called "differentiate",

```
(differentiate '(* (+ a b) (+ x y)) 'x)
```

should return

```
(+ (* (+ a b) (+ 1 0)) (* (+ x y) (+ 0 0))).
```

But it should be obvious that this expression can be greatly simplified to (+ a b), which is indeed the correct answer. You may want to use the simplifier given in chapter 3 of the notes in checking your answers.

Lisp provides a useful function called "trace" for checking and debugging functions you may write. It is an fsubr and takes a variable number of arguments which are names of defined functions. Then, whenever any of these functions are entered, Lisp will print the name of the function, the arguments which are being passed to the function, and a number indicating the depth to which the function is recursing. For example, if we had done (trace differentiate), then the previous example would have produced the following result:

```
(1 enter differentiate ((* (+ a b) (+ x y)) x))
  (2 enter differentiate ((+ x y) x))
    (3 enter differentiate (x x))
    (3 exit differentiate 1)
    (3 enter differentiate (y x))
    (3 exit differentiate 0)
  (2 exit differentiate (+ 1 0))
  (2 enter differentiate ((+ a b) x))
    (3 enter differentiate (a x))
    (3 exit differentiate 0)
    (3 enter differentiate (b x))
    (3 exit differentiate 0)
  (2 exit differentiate (+ 0 0))
(1 exit differentiate (+ (* (+ a b) (+ 1 0)) (* (+ x y) (+ 0 0))))
```

The fsubr "untrace" will discontinue tracing a given function, and if called with no arguments, will untrace all functions which are currently being traced.

If you should manage to complete any or all of problems 2 through 6, we would greatly appreciate your turning in a listing of your functions at the next session.

Lisp -- A Radical Introduction

Solutions to Worksheet #3

NOTES

A few things you should have observed from doing the exercises in the first problem:

a) Placing vertical bars around a symbol is equivalent to slashifying each of the characters in the symbol's print-name. The exception to this rule is that if a vertical bar or a slash is to appear in the name, it must still be slashified, i.e. preceded by a slash.

b) The functions "mapc" and "mapcar" are actually nothing more than fancy "do loops" -- we can easily define a "mapc" and a "mapcar" which take only two arguments:

```
(defun mapc (function arglist)
       (do ((args arglist (cdr args)))
           ((null args) arglist)
           (funcall function (car args))))

(defun mapcar (function arglist)
       (do ((args arglist (cdr args))
            (results nil (append results (funcall function (car args)))))
           ((null args) results)))
```

c) "print" is to "princ" as "explode" is to "explodec".

d) Functional definitions are attached to symbols via their property lists. Built-in functions will have either an "subr" or "fsubr" property which is the address corresponding to its location within the machine, and user-defined functions will have an "expr" or "fexpr" property which is the lambda expression we want to replace the symbol when it is called.

PROBLEM 2

This problem is solved most simply by exploding the print-names of each of the arguments into two lists, appending these lists together, and then imploding the result.

```
(defun concatenate (a b) (implode (append (explodec a) (explodec b))))
```

PROBLEM 3

Essentially, the strategy here is to create a new list by applying "rev2" to each of the elements within the given list, and then reversing this new list by applying the built-in Lisp function "reverse".

```
(defun rev2 (list)
       (cond ((atom list) list)
             (t (reverse (mapcar 'rev2 list)))))
```

PROBLEM 4

This function works by taking the cdr of the list it is given, sorting it, and then inserting the car of the list into this new sorted list. Note that the cdr must be sorted first before attempting to insert the car since "insert" assumes that its second argument is already sorted. Note also that if we replace the predicate "<" with "alphalessp", these functions can be used to sort lists of atoms.

```
(defun sort (list)
       (cond ((null list) nil)
             (t (insert (car list) (sort (cdr list))))))

(defun insert (atom sorted-list)
       (cond ((null sorted-list) (list atom))
             ((< atom (car sorted-list)) (cons atom sorted-list))
             (t (cons (car sorted-list) (insert atom (cdr sorted-list))))))
```

PROBLEM 5

Evaluating (who 'abraham) returned a list of abraham's grandchildren. It did this by appending together the "kids" properties of each of abraham's "kids".

PROBLEM 6

The following examples and the simple definitions of "mapc" and "mapcar" given above both make use of the function "funcall" which we will briefly introduce here. As its name may imply, the first argument to "funcall" is the name of a function to be called, and the subsequent arguments to "funcall" are arguments to be passed to the function being called. For example, (funcall 'car '(a b c)) is equivalent to saying (car '(a b c)).

In writing the differentiation function, it would be possible to include a test for each of the operations (addition, multiplication, or exponentiation) which our function can handle, and this is certainly a valid way of solving the problem. The method we have chosen, however, associates a separate differentiation function with each of the operations (by means of property lists). The simplifier which we've included here also makes use of this technique.

```
(mapc '(lambda (x y) (putprop x y 'diffn))
      '(+ * ^)
      '(diffplus difftimes diffexpt))

(defun diff (f x)
       (cond ((equal f x) 1)
             ((atom f) 0)
             (t (funcall (get (car f) 'diffn) f x))))

(defun diffplus (f x) (cons '+ (mapcar '(lambda (a) (diff a x)) (cdr f))))

(defun difftimes (f x)
       (prog (u v)
             (setq u (cadr f)
                   v (cond ((null (cdddr f)) (caddr f)) (t (cons '* (cddr f)))))
             (return (list '+ (list '* u (diff v x)) (list '* v (diff u x))))))
```

```
(defun diffexpt (f x)
       (prog (u v)
             (setq u (cadr f) v (caddr f))
             (return (list '+
                           (list '* v (list '^ u (list '- v 1)) (diff u x))
                           (list '* (list '^ u v) (list 'log u) (diff v x))))))

(mapc '(lambda (x y) (putprop x y 'simpfn))
      '(+ - * // ^ log)
      '(simplus simpminus simptimes simpquotient simpexpt simplog))

(defun evalp (exp)
       (and (member (car exp) '(+ - * // ^ log))
            (apply 'and (mapcar 'numberp (cdr exp)))))

(defun simplify (exp)
       (cond ((atom exp) exp)
             ((evalp exp) (eval exp))
             (t (funcall (get (car exp) 'simpfn) (mapcar 'simplify exp)))))

(defun simplus (exp)
       (delete 0 exp)
       (cond ((= (length exp) 2) (cadr exp)) (t exp)))

(defun simpminus (exp)
       (delete 0 exp)
       (cond ((= (length exp) 2) exp)
             (t (append (list '+ (cadr exp))
                        (mapcar '(lambda (x) (list '- x)) (cddr exp))))))

(defun simptimes (exp)
       (delete 1 exp)
       (cond ((member 0 exp) 0)
             ((= (length exp) 2) (cadr exp))
             (t (apply 'append
                       (mapcar '(lambda (x) (cond ((atom x) (list x))
                                                  ((equal (car x) '*) (cdr x))
                                                  (t (list x))))
                               exp)))))

(defun simpquotient (exp)
       (delete 1 exp)
       (cond ((equal (cadr exp) 0) 0)
             ((member 0 (cddr exp)) (error '|Division by zero.|))
             ((= (length exp) 2) (cadr exp))
             (t (append (list '* (cadr exp))
                        (mapcar '(lambda (x) (list '^ x -1)) (cddr exp))))))

(defun simpexpt (exp)
       (delete 1 exp)
       (cond ((equal (cadr exp) 0) 0)
             ((equal (cadr exp) 1) 1)
             ((= (length exp) 2) (cadr exp))
             (t exp)))

(defun simplog (exp) (cond ((eq (cadr exp) 'e) 1) (t exp)))
```