

T H E P - L I S P T U T O R I A L

by

Jeff Shrager

and

Steven Bagley

edited by Stewart Schiffman

Published by: Gnosis, Inc.  
4005 Chestnut St.  
Philadelphia, PA 19105  
(215) 387-1500



## **The P-LISP Tutorial — Table of Contents**

**Preface: Preface**

**Chapter 1: Getting Started**

**Chapter 2: LISTS, CAR, and CDR**

**Chapter 3: More Lists**

**Chapter 4: Atoms and Values**

**Chapter 5: Bag of Predicates**

**Chapter 6: Defining your own Functions**

**Chapter 7: Help Functions**

**Chapter 8: What is this thing called Lambda?**

**Chapter 9: The Conditional**

**Chapter 10: Simple Recursion**

**Chapter 11: Tracing a Function**

**Chapter 12: Lists as Trees**

**Chapter 13: Trees and Recursion**

**Chapter 14: A Style of Programming**

**Chapter 15: Scope Considerations**

**Chapter 16: A Catalog of Lisp Functions**

**Chapter 17: MAPS**

**Chapter 18: ISPLAY OGRAMMINGPRAY**

**Chapter 19: FEXPRs -- Unevaluating Functions**

**Chapter 20: EVAL and APPLY**

**Chapter 21: Properties and Lambda expressions**

**Chapter 22: Differentiating Polynomials**

**Chapter 23: Simplifying Polynomials**

**Chapter 24: Efficiency and Elimination of Recursion**

**Entire work copyright 1982, Steven Bagley and Jeff Shrager**

**Published by GNOSIS, a division of Pegasys Systems, Inc.**



**Look you lisp and wear strange suits.**

**--As You Like It (iv, 1, 34)**



## Preface

This book is a primer on LISP programming. It is written for any student wishing to gain a basic proficiency in LISP, regardless of his or her background in computing. In general, the level of discussion is appropriate for any high school student, college student or computer professional. A knowledge of elementary calculus will make understanding some of the later examples easier but by no means is required for the rest of the book.

### Why should you learn LISP?

LISP is important. LISP is one of the oldest languages still in active use. LISP was invented for and is still used [worshipped] by computer scientists interested in "Artificial Intelligence". AI, as it is called, is an area of active computer science research. For this work, LISP is indispensable.

LISP is simple. Many computer languages force the user to deal with messy details of the computer on which they are run. In LISP you don't worry about the mechanics of the computer. Also, the syntax, or format, of LISP expressions is regular and consistent.

LISP is fun. The types of problems usually dealt with in LISP often include games and puzzles. Also, LISP sessions are completely interactive. This interaction gives the user a greater sense of control over the machine, and makes the computer more of a "partner in thinking". Don't forget that for many years all computer systems were "batch", which meant that jobs had to be submitted on, horror of horrors, punched cards. The computer that P-LISP runs on is substantially more powerful, and vastly easier to use than most of those early machines.

### A Short History of LISP

LISP was developed in the late 1950's by John McCarthy at MIT to serve as an algebraic list processing language [LISP=LIST Processor] for work in the then new field of Artificial Intelligence. The first work on implementation began in 1958 and LISP 1 was born. A second version, called LISP 1.5, was completed in the next few years. LISP 1.5 is the precursor of most of the LISP systems in existence today. During the 1960's several other versions were developed across the country for various different machines. MACLISP from MIT, INTERLISP, formerly BBN LISP, and MTS LISP from the University of Michigan are three currently available. The dialect spoken in this book is P-LISP, for various microcomputers, written by Steven Cherry of PEGASYS SYSTEMS INC. Pegasys has now become GNOSIS, Inc.

### The Style of the Book

We believe learning should be fun and this book is written with that philosophy in mind. Thus, at times we resort to using "cute" examples to keep you from becoming bored with dry material.

We have tried to minimize any difficulties associated with some of the more abstract concepts in LISP by working up to them from elementary, concrete examples. We suggest that the reader carefully follow through all the examples presented. Access to your computer is desirable so that you may try your hand at LISP; nothing promotes learning like immediate feedback.

The chapters are quite short. It should be possible to read and comprehend several in one sitting. The entire book was not meant to be read in one sitting, so take your time.

This book was written by Jeff Shrager (currently at Carnegie Mellon University) and Steve Bagley (currently at MIT). It was originally written at the Moore School Computing Facility, in the School of Engineering and Applied Sciences, at the University of Pennsylvania, Philadelphia, PA.

### Typographic Conventions

Here are some conventions we will follow throughout this book. Lower case is used to indicate that a line of text is being typed by the user; similarly, upper case denotes lines typed by the LISP system.

### Overview

The format of the book is as follows. We start with a chapter of simple examples to get you comfortable with using the LISP system. We then move into several chapters which introduce the basic data structure and functions. After that, we tell you how to define and edit your own functions. We then introduce the concept of recursion, fundamental to LISP programming, and spend several chapters exploring different uses of recursion. The last section of the book consists of some chapters on advanced LISP techniques and several major applications.



As yet a child, nor yet a fool to fame,  
 I lisped in numbers, for the numbers came.  
 --Alexander Pope

## Chapter 1: GETTING STARTED

This chapter will provide you with some experience in using the P-LISP system. Its purpose is to help you become familiar with the basic operation of the language.

We assume that you are sitting in front of your computer, with P-LISP up and running (see the explanation of how to do this in the P-LISP USER'S GUIDE). You should see the following at the top of the screen:

```
GNOSIS, INC.  
P-LISP VER 3.0  
-----
```

COPYRIGHT 1981 BY STEVEN CHERRY  
 ALL RIGHTS RESERVED

```
:
```

When you see this display it means that you have successfully entered P-LISP. The ":" prompt that you see on the last line typed by the computer means that P-LISP is waiting for you to type something in. You may type in what you wish. After you hit the RETURN key P-LISP will evaluate your command and display the result. This process of read-evaluate-print result constitutes the core of the interactive LISP system. We will see more of READ-EVAL-PRINT much later on.

Note that if you hit RETURN several times, each time P-LISP will respond with the ":". You told it to do nothing, so it did nothing and then asked for another line of input.

If we type a number, then LISP will echo the number back. All of our inputs follow the ":" prompt; all of P-LISP's responses are preceded by two spaces.

```
:3  
  3  
:0  
  0  
:-2  
 -2
```

Let's try an example: adding up some numbers. To add numbers in P-LISP we use the ADD function. We add 1 and 2 by typing:

```
:(add 1 2)  
  3
```

Yeah! P-LISP can add. What actually happened? P-LISP typed the ":" and then we typed "(add 1 2)". Note several things:

- The word **ADD** and the numbers "1" and "2" are separated by spaces [blanks].
- We surrounded the expression with parentheses. Parentheses are an integral part of the LISP language, so you will soon learn to love parentheses [we hope].
- P-LISP responded immediately with the answer. LISP is an interactive system, and it will always display the answer immediately, unless you tell it otherwise. We will see later how to tell it otherwise.

Let's try some more addition.

```
:(add 1 2
;)
3
:(add 1 1 1)
3
:(add 1)
** ERROR! TOO FEW ARGS **
ADD :: NIL
+()
:(add)
** ERROR! TOO FEW ARGS **
ADD :: NIL
+()
```

Here we first try the same example but we have forgotten the closing parenthesis. P-LISP is waiting for that closing parenthesis so it comes back with a ":" prompt. We issue the closing parenthesis, and now P-LISP is happy, so it performs the addition. In general, you may spread the input across as many lines as you like--later this will be quite useful.

The next thing we tried was to add up more than two numbers at once. P-LISP did it correctly. In fact, P-LISP will add up any list of reasonable size using the **ADD** function.

The next line shows something a little funny. We asked P-LISP to add up one number. Since adding up just one number is not particularly meaningful or useful, P-LISP returns an error message that there are too few arguments. This is quite reasonable, since you usually want to add up at least two numbers.

Note that LISP now gives us a "+" instead of the usual ":". Don't worry about this for now, and just simply type "()". We will deal with this mode of operation later.

The last line shows what happens if you try to add up zero numbers: the same error again! Well, there is no penalty for mistakes [we won't tell].

The first thing in a pair of parentheses is the function name and the things after that are the argument names [thus the statement "TOO FEW ARGS" in the above error report]. This is very important, and these two words will be used throughout this book. In the first example above the function name is **ADD** and its arguments are "1" and "2". **ADD** is said to have two arguments in this example.

In addition to addition, P-LISP can perform multiplication. The name of the multiplication function is [surprise] **MULT**. Let's try it out!

```

(mult 2 3)
6
(mult 9 2)
18
(mult 1 2 3 4)
** ERROR :: TOO MANY ARGS **
MULT :: (2 3 4)
+)
(mult 1.2 4)
4.8
(mult 2 (add 1 2))
6

```

The first two examples reassure us that P-LISP can in fact multiply.

P-LISP can however multiply only two values. If you try and multiply more than two values, you will get a TOO MANY ARGS error. By the way, if you try and use MULT with zero or one argument, you will get a TOO FEW ARGS error.

The next example shows that P-LISP will deal with non-integers. Note that this would have given a BAD SYNTAX error under certain old versions of P-LISP, or without certain hardware. As we will see later, this does not matter a great deal since LISP's strength does not lie in arithmetic.

The last of the above lines is the most interesting. P-LISP tries to perform the MULT function but finds that in place of the second argument is a "subexpression". The multiplication operation is suspended while P-LISP evaluates that subexpression. The value of the subexpression "(add 1 2)" is, of course, 3. There is now a number to take the place of the subexpression so the multiplication can continue. P-LISP now effectively sees "(mult 2 3)" which it performs.

Since this type of operation is very common in LISP work, we are going to try some more examples like that last one. See if you can figure out what is happening in each expression.

```

(add (mult 3 4) (mult 2 6))
24
(mult (mult (add 1 0) (add 1 1)) (mult (add 1 1 1) (add 1 1 1)))
24
(mult 1 (mult 2 (mult 3 (mult 1 4)))))))))
24

```

Now for one more new concept: predicates. A predicate is a special kind of function that returns an answer of either true or false. In LISP true is represented as "T" and false is represented as "NIL". So, let's ask some questions:

```

(greater 3 4)
NIL
(greater 4 3)
T
(greater 100 -100)
T
(number 47)

```

```
T
(number 'letters)
NIL
(number 'seven)
NIL
(zero 0)
T
(zero (add 2 -1))
NIL
(zero (add 2 -2))
T
```

The predicate **GREATER** returns a true "T" if the numbers are in a strictly decreasing order, false "NIL" otherwise. The predicate **NUMBER** says "T" if the argument is a numeric atom, "NIL" otherwise. Obviously the string 'seven is a character atom (more on what that quote in front of it means later) and is not a number. Zero returns "T" if the argument evaluates to zero.

## Chapter 2: LISTS, CAR and CDR

We are now going to direct our attention towards the structure of data in the LISP language. All expressions in LISP are in the form of a list. Even functions that we will define in a later chapter will be in the form of a list. Lists are so important that the next several chapters will be devoted to developing your facility in using lists.

And now, meet the list.

A list is a linear arrangement of objects separated by blanks and surrounded by parentheses. The objects which make up a list are either "atoms" or other lists. An atom is the basic unit of data understood by the LISP language.

Here are some atoms:

```
carbon
eve
1
bananastand
```

Here are some lists:

```
(1 2 3 4)
((i hate) (peanut butter) (and jelly))
(you (walrus (hurt) the (one you) love))
(add 3 (mult 4 5))
(garbage (garbage) out)
(car ((in the garage) park))
(deeper and (deeper and (deeper and (deeper we went))))
```

Please note several things:

- Some of the atoms in the above lists are: "i", "()", "4", and "deeper". An atom is a word or number or the pair of parentheses "()" which will be referred to as "NIL".
- The parentheses in a list will always be balanced because every list is surrounded by a left and right parenthesis and the only things inside which have parentheses are other lists.
- The definition given above permits us to nest lists within other lists to any arbitrary depth.
- For clarity, we list here the elements of the third list above:

```
"you" is an atom.
"(walrus (hurt) the (one you) love)" is a list.
```

the parts of that list are:

```
"walrus" is an atom.
"(hurt)" is a list with one element: the atom "hurt".
"the" is an atom.
```

"(one you)" is a list with the elements:  
 "one" and "you", each of these is an atom.  
 "love" is an atom.

What does LISP do with lists? Well, whenever you type a list into LISP it tries to evaluate that list.

Rules for lists being evaluated:

- The first element of the list should be a LISP function [like ADD].
- The rest of the list should be the "arguments" to the LISP function, that is, it should contain the data to be acted upon.

Evaluation takes place if LISP can apply the function to the arguments.

Thus,

```
(add 1 2 3)
6
```

is a list which is evaluated and has its value printed.

If the first element is not a LISP function, then an error occurs:

```
((1 2 3 4)
  ** ERROR! BAD ATOMIC ARG **
  EVAL :: (1 2 3 4)
+())
NIL
```

What if we try to add all the numbers in a list?

```
((add (1 2 3 4))
  ** ERROR! BAD ATOMIC ARG **
  EVAL :: (1 2 3 4)
+())
NIL
```

Compare the expressions (ADD 1 2 3 4) and (ADD (1 2 3 4)). In the first one, the ADD function acts on a stream of atoms [not a list -- no surrounding parentheses] while in the second one ADD acts [or at least tries to act] on a list: (1 2 3 4). When LISP encounters this list, it suspends the addition operation and evaluates this list. Note that "1" is not a LISP function. [Remember, if LISP is trying to evaluate a list, the first element in the list had better be the name of a LISP function and the rest of the list had better be the arguments to that function or else TROUBLE!!]

Here, again, NIL ["()"] is used to get back to the normal LISP prompt ":",

We would like to be able to use list like "(a b c)", to represent data in LISP. Unfortunately LISP seems to want to evaluate everything that we enter. Since there is likely no "A" function, the evaluation of the list will cause an error. This leaves us in a bit of a quagmire!

Good fortune has fallen upon you! there is a way to stop LISP from trying to evaluate a list. The quote character `'` causes LISP to take the expression as written rather than to try to evaluate it.

```
;(do not (eat (anything) now))
  (DO NOT (EAT (ANYTHING) NOW))
;(mult (add 1 2) 4)
  (MULT (ADD 1 2) 4)
```

Let's introduce some LISP functions which manipulate lists. To manipulate involves taking apart, putting together, and checking the values of lists. The two functions CAR and CDR are used to get parts out of lists. The CAR function returns the first element in a list.

```
;(car '(1 2 3 4))
  1
;(car '((i hate) (peanut butter) (and jelly)))
  (I HATE)
;(car 1)
  ** ERROR: BAD ATOMIC ARG **
  CAR :: (1)
+()
  NIL
```

Note that the result of a CAR need not be an atom [in the second case above, it is a list of two atoms] but that CAR is only designed to take arguments which are lists, not atoms.

CDR [pronounced "could-er"] is the conceptual opposite of CAR in that the result of CDR is the "rest" of the list:

```
;(cdr '(1 2 3 4))
  (2 3 4)
;(cdr '(fun frog))
  (FROG)
;(cdr '(((three blind) mace)))
  (MACE)
;(cdr '(hello))
  NIL
;(cdr '())
  NIL
```

Like CAR, CDR is defined only to operate on lists. Unlike CAR, however, the value of CDR is ALWAYS a list. Note that the CDR of a list with only one element is an empty list [written as `()` or `NIL`].

We have, in the previous pages, listed the following seemingly contradictory characteristics of `NIL`:

- `NIL` is an atom.
- `NIL` is a list (as the result of the CDR operation).
- `NIL` means "false" in predicates.
- `NIL`, by name, means nothing.

NIL is certainly making a lot of trouble for such an empty concept. Why should we make so much ado about nothing? NIL is in fact the most important entity in the LISP language. It is both an atom and a list depending upon who is doing the asking. It can be returned by functions whose value is defined to be an atom, such as a predicate, or by functions whose value is defined to be a list, such as CDR. NIL is an empty list [a list with no elements]. The use of NIL will become clearer when we begin studying user defined functions in a later chapter.

Back to the business at hand: CAR and CDR.

We saw in the first chapter that subexpressions can be used in place of the arguments of any function. In the same way, the list processing functions can be combined to do various list operations.

```

:(cdr '(sand witch))
(WITCH)
:(cdr (cdr '(sand witch)))
NIL
:(cdr (cdr (cdr '(sand witch))))
NIL
:(car (cdr '(sand witch)))
WITCH
:(car (car (cdr '(() ((bozo) (no no)))))
(BOZO)
:(cdr (car (cdr '(() ((bozo) (no no)))))
((NO NO))
:(car (car (cdr (car (cdr '(() ((bozo) (no no)))))
NO
:(cdr (car '((car cdr) car)))
(CDR)
:(car '(add 1 2 3 4))
ADD
:(cdr '(add 1 2 3 4))
(1 2 3 4)

```

As we mentioned a little earlier in this chapter, the expressions that we are typing into LISP are lists in the same way that "(1 2 3 4)" is a list. Remember "functions" and "arguments"? Well, the CAR of an expression-list is its function name and the CDR of that expression-list is the list of the arguments to that function!

There are standard abbreviations for up to four successive applications of CAR/CDR combinations: take the letter "A" from every CAR and "D" from every CDR and place them next to each other sandwiched between a "C" and an "R". For example, the expression: (caddr anylist) is the same as the longer expression: (car (cdr (cdr anylist))). The above example:

```
:(cdr (car (cdr '(() ((bozo) (no no)))))
```

could have been written:

```
:(caddr '(() ((bozo) (no no)))
((NO NO))

```



I've got a little list...I've got a little list.

---The Mikado

### Chapter 3: MORE LISTS

We will now explore the domain of joining and extending lists. The most important LISP functions for joining lists are CONS and CONC. [These names stand for CONStruct and CONCatenate. They are a little more reasonable than CAR and CDR, but not much.]

Let's first play with the CONS function.

```

:(cons 'a '(b c))
(A B C)
:(cons '(a) '(b c))
((A) B C)
:(cons '() '(b c))
(NIL B C)
:(cons '(b c) '())
((B C))
:(cons '(a b) '(c d))
((A B) C D)
:(cons 'bacon '((lettuce) ((gazelle))))
(BACON (LETTUCE) ((GAZELLE)))
:(cons 'bacon '())
(BACON)

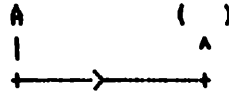
```

The explanation of CONS is a little tricky, so hang on. CONS takes its first argument [which may be either an atom or a list] and inserts it just after the first left parenthesis in the second argument. This second argument should be a list. CONS will actually connect things onto atoms as: "(cons 'a 'b)", but this creates a special form of list called a dotted pair. We will talk more about these later.

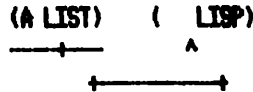
If you have a list of atoms, then you can use CONS to add another atom on the front of the list [as in the first example]. You can see that if we try to add an empty list [NIL or ()] to the front of the list, CONS will do it. If we try to add a list onto the front of a "()", then the "()" is treated as a list [just a set of balanced parentheses].

Here is a visualization of exactly what CONS will try to do:

```
(CONS 'A '())
```



```
(CONS '(A LIST) '(LISP))
```



CONS is very important. You should make sure you really understand how it works before proceeding.

Good. Well, the next magical function is called CONC. Again, let's just play around with CONC before discussing it.

```
:(conc '(ima list) '(ura list))
(IMA LIST URA LIST)
:(conc '((number one)) '(((number two))))
((NUMBER ONE) (NUMBER TWO)))
:(conc '(ready set go) '())
(READY SET GO)
:(conc '() '(go set down))
(GO SET DOWN)
:(conc '() '())
NIL
```

What does CONC do? [Can you answer that question now?] The CONC function joins two lists by sticking the first one onto the front of the second one and then removing one and only one pair of ")"(" from the middle. CONCing "(a)" with "(b)" will form "(a (b))". You then remove the ")"(" from the middle and you are left with the result "(a b)". Note that putting the lists together does not join "a" and "b". In other words, you don't get "(ab)". Both the arguments to and the result of a CONC are lists. The following shows what happens when you try to CONC atoms.

```
:(conc 'a '(b c))
** ERROR: BAD LIST ARG **
CONC :: ((QUOTE A) (QUOTE (B C)))
+()
NIL
:(conc '(b c) 'a)
** ERROR: BAD LIST ARG **
CONC :: ((QUOTE A))
+()
NIL
```

Moral of the story: CONS can deal with atoms, CONC can't.

And now the expression you've all been waiting for:

```
(cons (car '(i wanna go home)) (cdr '(i wanna go home)))
(i wanna go home)
```

Amazing. We took a list apart using CAR and CDR and then turned right around and put the list back together with CONS! Make sure you understand what is going on in the above example, making sure you can account for all the parentheses. Let's try using a CONC and a CDR for the CONS and the CAR. Note that we do not get back the list we started with.

```
(conc (cdr '(humpty dumpty)) (cdr '(humpty dumpty)))
(dumpty dumpty)
```

Using two CDR's we cannot put "(humpty dumpty)" back together again. This shouldn't be much of a surprise...we threw out the CAR of the list.

A quick note on the structure of lists. If a list contains another list as an element, then the inner list is said to be nested in the outer one. Also, it is often necessary to discuss "top-level" elements and "levels" of nesting. Here is a list with its "top-level" elements numbered:

```
(aka (googoo dada) waka)
-----
  1      2      3
```

Thus, there are 3 top level elements. The atoms "googoo" and "dada" are said to be more deeply nested; they are not on the top level.

What good are lists? Why would you want to use these CAR's, CDR's, CONS's and CONC's on lists? Answer! we can use lists to store data and the list provides us with a very flexible data structure. Let's spend some time investigating how lists can represent different kinds of things.

Suppose you have a bunch of friends and their phone numbers and you want to organize them. What is the important concept here? Each person will usually have just one phone number associated with him/her. Let's represent the pair <person, number> as a two element list: (person, number). Your phone book then becomes a list of two element lists. It might look like this:

```
((bill 1234567) (simon 5551212) (jane 2019999))
```

As a numerical example of data structure, consider a polynomial:

$12(x+2) + 17$

How can we represent this in LISP? There are lots of ways. We might use the built-in LISP functions for arithmetic operations to form an equivalent expression. The above polynomial is represented as:

We are going to return to polynomials of this type in later chapters and show you how to manipulate them in meaningful ways. Before that, however, we are going to have to see some more of the LISP language.

The technical term, Atom, marks sufficiently the nature of the opinion. According to this theory, the world consists of a collection of simple particles....

--- Whewell, History of Inductive Science (1857)

## Chapter 4: ATOMS AND VALUES

In the previous chapters we discussed lists of objects. These objects could have been either atoms or other lists. The exact meaning of atom was side-stepped. Here we will look a little deeper into what an atom is and how they hold information.

Let's go back and repeat the definition of an atom given in chapter 2:

An atom is a word or a number or the pair of parentheses "()" which we will call "NIL".

So, by that description, all of the following are atoms:

```
hello
31415
()
car
axe
0
stephen
anatom
12345
```

In fact, there are a few more things that we can use as atoms also. The rules for creating atoms are, to be exact:

- An atom can be any integer between -32767 and 32767. This is called a numeric atom.
- A non-numeric atom can be any name made up of letters and/or numbers. There is no limit on the length of this. The only restriction is that the first character must be a letter, not a number. This is called an alphabetic atom.
- The form NIL "()" can be an atom.
- Alphabetic atoms can have some funny characters in them [like "\*" and "+"] but special LISP characters can't be used in atom names. This should be clear by now. The characters "(" and ")" and "\"" would simply confuse LISP if you tried to use them in atom names. In general, we avoid using anything other than the letters "A" through "Z" and the numbers "0" through "9" in atom names.

Using the above rules, these are not atoms:

```
456test
a sentence like this is not an atom
some'stuff
this(isn't)(one(either
(1 2 3 4) <-- this one's a list, remember?
```

As we usually do, we'll see what happens when atoms are given to LISP to evaluate:

```

:anatom
  ** ERROR: UNDEFINED ATOM **
  EVAL :: ANATOM
+()
  NIL
:t
  T
:nil
  NIL
:()
  NIL
:567
  567

```

That seemed to work alright, except for the first one. What happened? It looks like some atoms are "DEFINED" according to LISP and some aren't.

Atoms have values. Some atoms have values that are automatically set by LISP when you start it. Others need to be given values by you, the user, when you want to do something. When an atom is typed into LISP it is evaluated just like a list except that instead of executing a function, the result of the evaluation is the value of that atom.

We can see the types of atoms mentioned in the previous paragraph used in the example above. The atoms "T" and "NIL" seem to already have values in LISP. The value of "T" is "T", the value of "NIL" is "NIL" or "()" which, as we've said over and over, is the same thing. The atom 567 also has a value. In fact, all numeric atoms have values that are the numbers they represent. Numbers and those special atoms are called "self-defining" atoms.

Okay, then how do we define those atoms that aren't self-defining? There's a way to do that too! [There's a way to do most everything in LISP.]

Watch this:

```

:undefined-atom
  ** ERROR: UNDEFINED ATOM **
  EVAL :: UNDEFINED-ATOM
+()
  NIL
:(setq undefined-atom 5)
  5
:undefined-atom
  5
:(setq new-atom undefined-atom)
  5
:new-atom
  5
:(setq another-atom yet-another-atom)
  ** ERROR: UNDEFINED ATOM **
  EVAL :: YET-ANOTHER-ATOM
+()

```

**NIL**

The first example just shows that, in fact, the atom of interest is undefined. Let's define it.

The SETQ function takes the name of an atom and the value to "assign" to that atom. It puts that value into the named atom and then, voila, instant definition! Note that the value returned by SETQ is the same as the value of the second argument. You will find that all LISP expressions return a value of some kind.

Note that, in the fourth example, the atom now properly defined has a value that can be used to assign to other atoms. An error will occur if you try to assign the value of an undefined atom in a SETQ operation.

As with a list, if we want to tell LISP not to try to evaluate an atom, you can simply put a single quote before it!

```

the-value
  ** ERROR: UNDEFINED ATOM **
EVAL :: THE-VALUE
+()
  NIL
(setq atom1 'the-value)
  THE-VALUE
(atom1
  THE-VALUE
(cons atom1 '(is the value of atom1))
  (THE-VALUE IS THE VALUE OF ATOM1)
(setq atom2 (cons atom1 '(is the value of atom1)))
  (THE-VALUE IS THE VALUE OF ATOM1)

```

Above, we have used the value of atom1 and the CONS function to create a list made up of the value of atom1 and some other atoms. You should be very careful about the placement of quotes at all times. A quote is used when you mean "the expression itself" rather than the result of evaluation of the expression.

In the last example above the result of the CONS operation is SETQed into a new atom "atom2".

You will find that one of the best uses of SETQ is to save you from having to type the same list over and over and over again. We can use atom values in our examples to save our typing also. This is one of the examples from chapter 2 redone with value atoms and SETQ's:

```

(setq sandwich '(sand witch))
  (SAND WITCH)
(cdr sandwich)
  (WITCH)
(cdr (cdr (cdr sandwich)))
  NIL
(car (cdr sandwich))
  WITCH
(car (car (cdr (setq clowny '(() (bozo) (no no))))))

```

```

(BOZO)
:(cdr (car (cdr clowny)))
  ((no no))
:(car (car (cdr (car (cdr clowny)))))
  NO

```

You should go back to the example in the previous chapter and carefully study what was done in order to use value-atoms.

Let's not lose sight of the evaluation process in all this mumbo-jumbo. Remember that the arguments of a function are evaluated first and the results replaced for that position in the expression. Using this rule, let's go through the evaluation of the last expression above:

We start out with:

```
(car (car (cdr (car (cdr clowny)))))
```

First, "clowny" is evaluated as a list that was SETQed to it previously giving us:

```
(car (car (cdr (car (cdr '(( ) ((bozo) (no no)))))))
```

[The underlined portion is the SETQed text].

The various CARs and CDRs are evaluated giving:

```

(car (car (cdr (car '(((bozo) (no no))))))
(car (car (cdr '((bozo) (no no)))))
(car (car '((no no))))
(car '(no no))
no

```

This process is the most important thing that you need to know and it is assumed in all our discussions.



## Chapter 5: BAG OF PREDICATES

Congratulations. You have now mastered the basic concepts of atoms and lists. It is the purpose of this chapter to add to your "vocabulary" of LISP functions. You will need a few more in order to do any real work. All of the new functions will be predicates. Remember that predicates ask questions about data and always return T for true or NIL for false.

The first function is a predicate that asks if an expression is an atom.

```
(atom 'bomb)
T
(setq bomb 'kyag)
KYAG
(atom bomb)
T
(setq nixon '(i am not a cook))
(I AM NOT A COOK)
(atom nixon)
NIL
(atom '((and) (eve)))
NIL
(atom (car nixon))
T
(atom ())
T
```

As you will later see, to be able to test for atom-icity is very important. [Is atom-icity a word? Probably not, but it is an atom.]

Another nice testing function is NULL. NULL says T if and only if its argument is NIL. Now we have a way to test for NIL-icity [sorry].

```
(null bomb)
NIL
(null (setq alist '(let them eat cake)))
NIL
(null nil)
T
(null ())
T
(null t)
NIL
(null (null t))
T
(null (null nil))
NIL
(null (car ()))
T
(null (cdr ()))
T
(null berry-bush)
```

```

** ERROR: UNDEFINED ATOM **
EVAL :: BERRY-BUSH
+()
NIL

```

That last one is just to see if you're still awake.

It would be reasonable if you could test for the equality of two expressions. Since LISP is always in a reasonable mood, there is such a predicate: EQUAL. EQUAL returns a T if and only if its arguments represent the same LISP expression; the arguments may be atoms or lists, but there may only be two such expressions.

```

(equal '(a) '(a))
T
(equal 1 (add 1 1))
NIL
(equal (car '((a deep) list)) '(a deep))
T

```

Using the NULL and EQUAL predicates, we can make our own "not-equal" function.

```

(not-equal 1 (add 1 1))
T

```

that is, 1 does not equal 1 plus 1. LISP has a function called NOT which does exactly the same thing as NULL. It would be preferable to use NOT here because it makes a little more sense if you read it out loud.

More equality:

```

(equal 'head (car '(head for the nils)))
T
(equal 'solipsist)
** ERROR: TOO FEW ARGUMENTS **
EQUAL :: NIL
+()
NIL
(equal 'three 'for 'all)
** ERROR: TOO MANY ARGUMENTS **
EQUAL :: ((QUOTE FOR) (QUOTE ALL))
+()
NIL
(equal t (equal 4 (sub 11 7)))
T

```

We pulled a fast one in that last example: SUB is a new function. No problem, really. SUB does subtraction a la grade school. Play with it to make sure it does its homework.

You jig, you amble and you lisp, and nick-name God's creatures.  
 ---Hamlet (iii, 1, 151)

## Chapter 6: DEFINING YOUR OWN FUNCTIONS

Up to this point in the discussion, LISP has had a monopoly on functions; we have been forced to use the ones supplied by LISP. If we were limited to these, LISP wouldn't be much fun. What we are leading up to is that you, the user, are able to define, use, and even modify your own functions.

Suppose that you are one of those select people who do not like the name CAR. Well, we are going to define a function called FIRST which does exactly the same thing as CAR. Let's go into LISP.

```
:(define (first (lambda (l)
:   (car l))))
  FIRST
:(first '(this had better work))
  THIS
:(first '(long live define))
  LONG
```

The explanation: we used the function DEFINE to set up our FIRST function. DEFINE takes as its argument a list containing the function definition. Note that we took more than one line to enter the function definition. This will generally be the case. LISP will wait to see a matched set of parentheses. If you are not careful with the parentheses, you may have to do a lot of retyping. We can't explain everything about the form of the function definition at this point in the book. Suffice it to say that you must enter your functions in the above form. The first element in the function definition list is the name of the function you are defining, in this case, FIRST. The names that may be used for a function are the same as those for an alphabetic atom name.

The next thing in the function definition is a list whose first element is LAMBDA. Let's skip over the LAMBDA for the moment.

Following the LAMBDA is a list of "formal arguments" for your function. In our example, this is the list: "(l)". We have to make explicit to LISP the number of arguments our function will have. We have already seen LISP functions which take one, two, or sometimes an indefinite number of arguments. If our list had been (l1 l2) then we would be defining a function with exactly two arguments. Here, however, we only want one argument. Remember, we are defining a function to behave like CAR, which has only one argument.

After the list of arguments comes an expression whose value will be returned as the value of the function. In this case we want a CAR to be the result. Note that we use the formal arguments in this expression. This will be explained in more detail later.

The value of the LISP function DEFINE is the name of the function being defined.

Note that we called the list a list of "formal arguments". What are "formal arguments"? Well, we would like to be able to enter an expression like the following:

```
(first x)
```

where *x* is some pre-defined list. The name we use in the argument list in the function definition will, when the above expression is evaluated, take on the value of the list *X*. The name we use in our function, however, only serves as a place holder for the value of the actual argument. Note that here the actual argument is *X*. The formal parameter, which in the *FIRST* definition is *L*, takes on the value when *FIRST* is evaluated. Be careful here, because the value of *L* will revert to whatever value [or lack thereof] it had previously after the function had been evaluated.

```
:(first '(a lucky star))
A
:
** ERROR: UNDEFINED ATOM **
EVAL :: L
+()
NIL
```

We know that *L* had a value inside the function, ~~because~~ the function executed correctly. Once the function finishes, *L* goes away. Poof!

Let's try our hand at another function definition.

```
:(define (second (lambda (alist)
: (car (cdr alist))))
SECOND
:(second '(can you say the word fun))
YOU
:alist
** ERROR: UNDEFINED ATOM **
EVAL :: ALIST
+()
NIL
```

This dialogue shows the same characteristic behavior of formal arguments! they clean up after themselves. That is, once the function has been terminated, the values of the formal arguments are no longer available.

Now, let's deal with functions with two arguments. Let's define our own *EQUAL* function.

```
:(define (same (lambda (list1 list2)
: (equal list1 list2))))
SAME
:(same '(testing) 'testing)
NIL
:(same '(equal) list2)
** ERROR: UNDEFINED ATOM **
EVAL :: LIST2
+()
NIL
:(setq list2 '(invisible))
```

```

(INVISIBLE)
:(same 'clone 'clone)
T
:list2
(INVISIBLE)

```

Note that not only can't we get to the formal arguments after the function is finished, but if we have another object with the same name as a formal argument then that variable keeps its value even though the formal argument had a different value! That is both very important and very confusing. You may want to re-read this explanation and then try some of your own examples so that you develop a "feel" for the operation of the formal arguments.

Suppose we want a function that will return the last element of a list, sort of the opposite of CAR. Well, first we will need a LISP function called REVERSE. REVERSE will return a list with all of its top-level elements reversed. Let's make sure REVERSE works.

```

:(reverse '(p l e h))
(H E L P)
:(reverse '(s d r a w k c a b))
(B A C K W A R D S)
:(reverse '((a b) c d (e f)))
((E F) D C (A B))

```

Now we get to our function LAST. How do we go about getting the last element of a list. Well, now that we know about REVERSE, we can reverse the list, and then take the CAR. Let's try it.

```

:(define (last (lambda (zzz)
: (car (reverse zzz))))
LAST
:((last '(now is the time))
TIME
:(last '())
(NIL)

```

By this time you might be able to say to yourself, "So what?" Why bother defining a trivial function like LAST when you could just type out "(car (reverse...))"? The answer to this is two-fold. First, the functions that you define won't be so trivial later on. The work done in a single function is almost unlimited. We are just using simple examples at this point.

The second reason for learning how to define simplistic functions is more subtle. Remember that we mentioned the NOT function in chapter 5? It did exactly the same things as NULL. In fact we might have defined NOT [if it were not already there] by typing:

```

:(define (not (lambda (a)
: (null a)))
NOT

```

The reason for having both NULL and NOT is that in some cases it makes more sense to the programmer to envision a NOT than a NULL. Go back over the example in chapter 5 and you'll see this vividly.

Along the same lines as NOT and NULL, suppose that we were using a list to hold the names of our friends in the form:

(firstname middlename lastname)

We could then define functions called FIRSTNAME, MIDDLENAME, and LASTNAME to access the parts of the list. They would represent CAR, CADR, and CADDR respectively. It makes more sense to ask for "(middlename friend)" than it does to ask for "(cadr friend)".

Using simple defined functions in this way helps us to organize our own thoughts when designing a program and also helps others when they try to read our work. Applying meaningful names to simple things is one very important use of DEFINE.

I get by with a little help from my friends.  
--The Beatles

## Chapter 7: HELP FUNCTIONS

The functions we defined in preceding chapters used functions such as CAR and CDR. These functions are called BUILT-INS. A built-in function is nothing mysterious. All the built-in functions are just the same as your own functions (like FIRST). In fact, once you define a function, it becomes built-in until you leave LISP. The only difference between built-ins that you build in and those that are built in by P-LISP is that the latter are defined automatically each time you enter P-LISP.

A built-in that you build into LISP (like FIRST) is called a HELP function. We call these help-functions because we can use them to write other functions that do bigger and better things. [We could use them to do smaller and worse things but that isn't any fun.]

Let's write a function called ENDS which takes the first and last elements from a list and returns them as a new list. To begin with, let's redefine FIRST and LAST, since, unless you saved your workspace the last time, they are not there any more.

```
{(define (first (lambda (l)
:   (car l) )))
FIRST
(define (last (lambda (l)
:   (car (reverse l)) )))
LAST
```

Okay, now we define ENDS which uses FIRST and LAST as help-functions.

```
{(define (ends (lambda (l)
:   (conc (first l) (last l)) )))
ENDS
```

and we'll try it out!

```
{(setq inputlist '(grateful are those who are not dead))
(GRATEFUL ARE THOSE WHO ARE NOT DEAD)
(ends inputlist)
** ERROR: BAD LIST ARG **
EVAL :: ((FIRST L) (LAST L))
+1
(GRATEFUL ARE THOSE WHO ARE NOT DEAD)
+(first l)
GRATEFUL
+(last l)
DEAD
+()
NIL
{(first l)
** ERROR: UNDEFINED ATOM **
EVAL :: L
```

```

+()
  NIL

```

When LISP comes back with the "+" prompt, then the function we are executing [which was ENDS] is "suspended". This means that it has been stopped in mid-evaluation due to some error (or because you hit control C). At this time, the formal argument "1" has the value intact. We can look at its value and also use it in other functions while ENDS is still suspended. You know ENDS is still suspended because you still get the "+" prompt. After you type the "()", the suspension is cleared, and the formal argument is gone.

Let's look at the error report to see if we can find the problem with ENDS. Both FIRST and LAST successfully return values, so they are not the problem. AHA!! CONC takes as arguments two lists, not two atoms! Watch!

```

:(conc 'grateful 'dead)
  ** ERROR! BAD LIST ARG **
  EVAL :: ((QUOTE GRATEFUL) (QUOTE DEAD))
+()
  NIL

```

Looks like we'll have to figure out some way of making a list out of the result of FIRST and LAST.

How can we make an atom into a list? Think about what CONC does to the parentheses and about what we want the result to look like. Given "anatom" we want to see "(anatom)". We learned that the CONS function puts something into the beginning of a list. We can simply use CONS to stick the atom into a NIL list. Do you see that this will give us the result that we need? If not, try it in LISP.

Now that we have figured out what the problem was, how do we fix up poor old ENDS? Why not write another help-function called MAKELIST that puts parentheses around the result of FIRST and LAST for us!

```

:(define (makelist (lambda (atom)
  :   (cons atom nil) )))
  MAKELIST

```

And, try it out!

```

:(makelist 'deadhead)
  (DEADHEAD)

```

Okay, that looks good. Now all we have to do is change ENDS to use MAKELIST as a help-function. [Note the proliferation of help-functions.]

```

:(define (ends (lambda (l)
  :   (conc (makelist (first l)) (makelist (last l))))))
  ENDS
:(ends '(i cant believe i ate the whole think))
  (I THINK)
:(ends inputlist)
  (GRATEFUL DEAD)

```



How about that! It worked! It should be pointed out that P-LISP already has a function called LIST which does exactly what MAKELIST does. We did it ourselves just for practice.

A suggestion: go back through what we just did. Type it all into the computer and carefully follow every step. We'll talk more about figuring out what happens when a function goes wrong (a process called debugging) and about changing help-functions (a process called editing) in later chapters. For now, just understand what went wrong and how we fixed it.

The important concept in this chapter was the use of help-functions to make the job of other functions easier. Never be afraid to write a help function to perform some little task for you. As with the simple function definitions in the last chapter, they can help you to organize your thoughts by naming menial jobs into understandable parts.



Beside, 'tis known he could speak Greek  
 As naturally as pigs squeak.  
 --Samuel Butler

## Chapter 8: WHAT IS THIS THING CALLED LAMBDA?

All the functions that were defined previously had this LISP thing called **LAMBDA** in them. We left this without explanation up to now. Here we'll take a deeper look at the **LAMBDA** function and what it does.

Functions are always defined in the following form:

```
(DEFINE (name (LAMBDA (formal arguments)
                    function-expression )))
```

"Name" is the name of the function being defined. This shouldn't be any trouble.

"Function-expression" is simply the expression that will be evaluated when we invoke the named function. We've been here before, right? When you type "(name ...)" LISP evaluates: "(function-expression)". That was how we got **FIRST** to do a **CAR** function. Whenever "(first...)" was entered, LISP replaced it with its defined function-expression: "(car ...)".

"Formal arguments" serve to hold the place for the actual value(s) which will be inserted when the function is evaluated. As you will recall from the previous chapter, we had a formal argument [L] in the **FIRST** function. We couldn't get the value of L outside of the function. The "function-expression" can use L, but we can't! Why is that?

Let's look again at the function **FIRST**:

```
(define (first lambda (l)
          (car l) )))
```

Now, let's invoke **FIRST**:

```
((first '(a b c))
 A
```

This is the same result we would have gotten if we had instead done:

```
((setq l '(a b c))
 (A B C)
 (car l)
 A
```

except that if we had done that then the value of L would still have been available after the **CAR** operation. It seems that the formal arguments got **SETQ**ed to match the supplied arguments when the function was invoked. The only difference is that after the function finishes (i.e. the function-expression is done evaluating) the values get **unSETQ**ed.

If the function has an error that causes an interrupt [you get an error message and get a "+" prompt] then it hasn't finished evaluation yet and the values assigned to the formal arguments are still there. That is why we can look at them when we have the "+" prompt. When we tell LISP to terminate the function that was suspended by entering a NIL to the "+" then it terminates the function and, POOF, the values that were in our formal arguments are gone.

To be a little more specific, the supplied arguments in the calling expression are matched with the formal arguments. Then the values of the supplied arguments are inserted into the function expression where their associated formal arguments were.

Watch LISP evaluate FIRST:

we type: `"(first '(a b c))"`

`(LAMBDA (l) (car (l)))`

so the expansion becomes:

`(car '(a b c)) = a` (our result)

which is subsequently evaluated in place of the original expression.

Now let's discuss some variations on this theme.

If there had been two variables in the formal argument list `["(l m)"]` for example] then we would have had to put two variables in the supplied arguments part of the expression. It should be pointed out that the names in the formal argument list (also known as the lambda list) are completely arbitrary. "l" and "m" might just as easily have been "fred" and "moose", as long as we did the same in the function-expression also.

```
:(define (merge (lambda (l m)
:  (conc l m) )))
MERGE
:(merge '(a b c) '(d e f))
(A B C D E F)
```

The first supplied argument would get bound [that's short for connected] to the first name "l" and the second would get bound to the name "m".

```
( (a b c) (d e f) )
  ↓      ↓
(  l      m  )
```

Let's try not giving LISP anything to connect to the formal argument:

```
:(first)
** ERROR: TOO FEW ARGS **
EVAL :: NIL
```

+()  
NIL

Oops! Well, that should have been expected. FIRST had one formal argument and we supplied none, so it told us that we had too few. Now the error reports should be slightly more meaningful to you.

Can you figure out what would happen if we had typed: "(first '(a b c) '(d e f))"? Try it.

The process of assigning the values of the supplied arguments to the formal arguments is called Lambda-Binding. The arguments in the "lambda-list" [the formal arguments] are called locals. They are "local" to the function in which they are bound in that when that function ends the binding comes apart and L [in the case of FIRST] no longer has the value it had inside the function. That is why we can't see the value in L after the function has ended.

The process of Lambda Binding is the main driving force behind LISP. We will discuss its mechanisms in detail later on. For the time being it is important that you understand what it appears to do to variables.



If you think being massaged by one person is great, then wait until you try two.  
-- The Massage Book

## Chapter 9: THE CONDITIONAL

This chapter deals with conditional expressions in LISP. Conditions are used commonly in English: "If you're a bad boy then you'll be sent to bed without dinner".

The conditional above has two parts: a test, "you're a bad boy", and a statement which will result if the test is true, "you'll be sent to bed without dinner".

Here is a set of examples in LISP:

```
(cond (t '(hi there)) (nil '(their high)))
      (HI THERE)
(cond (nil '(their high)) (t '(hi there)))
      (HI THERE)
(cond (t '(their high)) (t '(hi there)))
      (THEIR HIGH)
```

COND takes as its arguments a set of lists. Suppose, for the sake of explanation, we use:

```
(cond list1 list2)
```

COND will evaluate the CAR of list1. If the result is not a NIL, then COND will return evaluate the remaining things in the list, and will return the value of the last thing it evaluated. If the result is NIL, COND will go on and do the same thing for list2.

In the first example above, the first list is: "(t '(hi there))". The CAR of the expression is "T", which evaluates to "T". Following the process described above, COND notes that the value T does not equal NIL, and therefore evaluates the rest of the list. Its value is "(hi there)" which is what is returned.

The second example shows the case where the first list is not evaluated but the second is. The CAR of the first list evaluates to NIL so that list is skipped. The CAR of the second list is T, so the rest is evaluated. Not too bad so far, eh?

If both the first and the second CAR's are true, then because COND starts at the beginning and works its way down, since the CAR of the first list evaluates to "T", COND will never get to the second list. The second list will never be evaluated!!

When COND can't find a list with a non-NIL CAR, then it returns a NIL:

```
(cond (nil '(falsehood)) (nil '(falsetto)))
      NIL
```

COND will work for any number of lists, not just two. Watch:

```
(cond (nil '(door #1)) (nil '(door #2)) (t '(door #5)))
      (DOOR #5)
```

Okay, the party is over, and now we put COND to work. Above, we used the values of T and NIL, because they evaluate to themselves. However, COND becomes a much more powerful tool when predicates are used as the first elements of the conditional lists.

The definition of the absolute value function can be described as:

"If the number is positive, then return the number. Otherwise, it returns the number's negative." Note that this successfully "catches" zero, since the negative of zero is still zero.

The equivalent LISP function can be written:

```
(define (abs (lambda (n)
:      (cond
:        ((greater n 0) n)
:        (t (mult n -1))) )))
ABS
:(abs 3)
3
:(abs -453)
453
:(abs 0)
0
```

Here is a non-numerical function which uses COND. Make sure you understand the evaluation of the function.

```
(define (make-a-list (lambda (a-list)
:      (cond
:        ((atom a-list) (cons a-list nil))
:        (t a-list))) )))
MAKE-A-LIST
:(make-a-list 'happy)
(HAPPY)
:(make-a-list '(lisp lisp lisp))
(LISP LISP LISP)
```



O! Call back yesterday, bid time return!  
 --Richard II (III, ii, 69)

## Chapter 10: SIMPLE RECURSION

Recursion happens when a program calls itself as a help function. How can a function be defined in terms of itself? That sounds like a circular definition!

Recursion avoids circularity by defining the function in terms of simpler cases of itself. If we keep using the function on simpler cases, then eventually the function will get to a simple enough case so that it knows the answer without having to recur.

Let's try a simple example: a program, called RECITE, to print out all the elements in a list, one element per line of output. We will do this by having our function print out the CAR of the list using the built-in LISP function PRINT and then call itself [recur] with the CDR of the list. Notice that since we are going to pass the CDR of the list, the list must get smaller with each recursive call.

Recursion is useless unless we can make it stop. So RECITE should do the following: if its argument is NIL, then it should return a NIL. This type of test is called a termination condition. We need it to keep LISP from running away. If its argument is not a NIL, then print the CAR of the argument and call RECITE again with the CDR. Here is RECITE:

```
(define (recite (lambda (stuff)
:   (cond ((null stuff) ())
:         (t (print (car stuff))
:             (recite (cdr stuff)) )
:   )))
RECITE
(recite '(this is a test list))
THIS
IS
A
TEST
LIST
NIL
```

When STUFF is NIL, the COND evaluates "(null stuff)" to T and evaluates the "()" as dictated by COND. Otherwise it prints the CAR of the list and calls RECITE binding the CDR of STUFF to the new STUFF. Notice that it does not replace the value of STUFF but simply binds a new local value to it. When that particular call terminates, the previous value of STUFF will return.

The NIL displayed at the end is not printed by the PRINT function. Rather, it is the value returned by the RECITE function. It came from the succession of recursive function terminations. When the function we started off with terminates, it prints out its value because there is no "caller" to return to other than the user.

What would have happened if we had left out the termination condition? Answer: no end in sight. Watch!

The list of NILs in the above execution will go on forever. We've cut it off at seven in order to preserve our forests.

```

(define (member (lambda (a l)
:      (cond
:      ((null l) nil)
:      ((eq a (car l)) t)
:      (t (member a (cdr l))) )))
MEMBER
(member 'man '(union man))
T
(member 'snurd '(elmer snerd))
NIL
(member 'a '((a b) c d))
NIL

```

Here, we first test to see if L is an empty list, because if it is then we need to search no further. We then compare the specified atom A with the first element of list L. If they are equal then we have a match and the value of T is returned. Otherwise, we try again, looking for the atom in the CDR [rest] of the list. Note that this process is guaranteed to terminate as the function either returns a value or tries again with a shorter list. A list can only contain a finite number of elements so that after a maximum number of calls equal to the number of top-level elements in the initial list, we must reach an answer.

Misery still delights to trace  
 Its semblance in anothers case  
 --William Cowper

### Chapter 11: TRACING A FUNCTION

In this chapter, we are going to follow the MEMBER function from the previous chapter, using a facility in the LISP system called TRACE. The LISP TRACE function will tell us who calls who and what is returned. When you see "-->>", it means that the function is being called. When you see "<<--" that indicates that the function is returning. Follow through these examples by hand and watch what's happening. [Note that you get an extra set of parentheses around the arguments in the "-->>" trace. Be careful of these!]

```
:(trace member)
MEMBER
:(member 'arm '(head leg arm foot))
-->> MEMBER :: (ARM (HEAD LEG ARM FOOT))
-->> MEMBER :: (ARM (LEG ARM FOOT))
-->> MEMBER :: (ARM (ARM FOOT))
<<-- MEMBER :: T
<<-- MEMBER :: T
<<-- MEMBER :: T
T
```

Note that the "T" result is passed back through each level of recursive call. It isn't just popped right back up to the top from the last call [the last "-->>"].

Let's try one that fails [returns NIL]:

```
:(member 'hand '(arm head leg foot))
-->> MEMBER :: (HAND (ARM HEAD LEG FOOT))
-->> MEMBER :: (HAND (HEAD LEG FOOT))
-->> MEMBER :: (HAND (LEG FOOT))
-->> MEMBER :: (HAND (FOOT))
-->> MEMBER :: (HAND NIL)
<<-- MEMBER :: NIL
<<-- MEMBER :: NIL
<<-- MEMBER :: NIL
<<-- MEMBER :: NIL
<<-- MEMBER :: NIL
NIL
```

The same returning sequence happens with the NIL. In fact, the same type of thing will always happen in LISP --- called functions return values to the routine that called them, never back to the user directly. We saw this in the ENDS example and it also applies to recursions.

The opposite of TRACE is UNTRACE:

```
:(untrace recite)
NIL
```

If you forget to **UNTRACE** your functions they will keep tracing themselves until you either restart LISP, or shut off your computer.

If you wish to turn off tracing of all your functions at once, simply type **(UNTRACE)**.

```
:(UNTRACE)
```

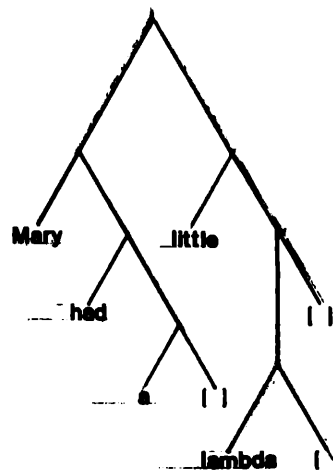
```
  NIL
```

A fool sees not the same tree that a wise man sees.  
--William Blake

## Chapter 12: LISTS AS TREES

There is a way of thinking about recursive functions that may help to clarify the method. This brief chapter will try to explain the simple LISP functions in terms of trees. We will then use the tree representation in later chapters to describe the action of recursive functions.

A neat way to look at what a deeply embedded list looks like is to think of it as a tree. A tree, as the name implies is a structure with a root, branches and leaves. This is a tree!



The above tree is the representation of the list!

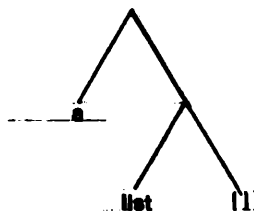
`((mary (had a)) (little (lambda)))`

Each node of the tree indicates the starting point of a pair of elements. The root [first node] indicates the starting point of the main list. The leaves of the tree are the atoms in the list.

Notice that each node of the tree branches two ways. This type of tree is called a "binary" tree because of this two way branching.

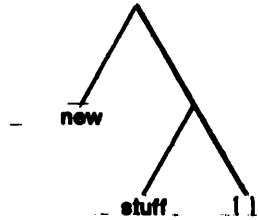
Let's take a look at a simpler tree!

(a list)



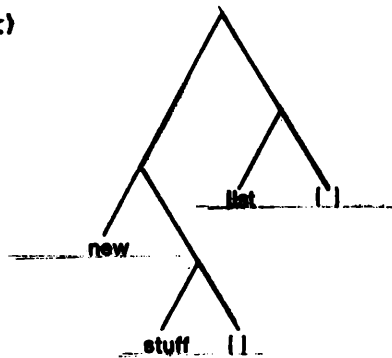
This tree has two nodes and four branches. There are atoms [NIL is an atom too!] at the ends of the lowest branches.

(new stuff)



It looks the same, Now, let's insert the second tree into the first, replacing "(new stuff)" for "a". The resulting tree looks like!

((new stuff) list)



When we look at the CAR of the above tree we are looking at!

(new stuff)

CAR can be thought of as returning all the stuff hanging on the left branch. CDR, as you might expect, returns the stuff on the right branch,

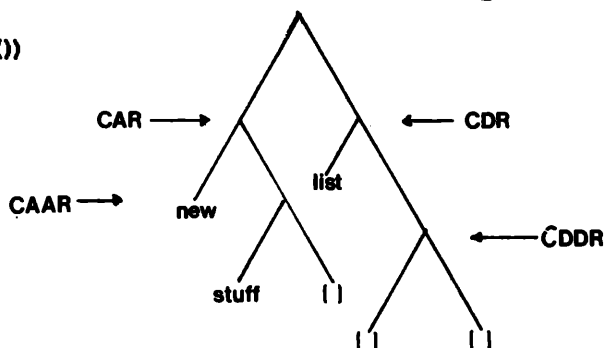
(list)



which is why there are still a set of parentheses around it.

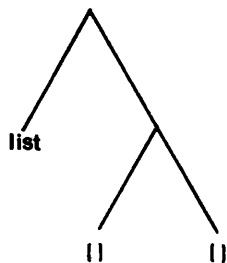
Note that all the trees eventually terminate in NIL ["()"]. Recall that if you keep taking the CDR of a list you will eventually hit a NIL! The reason for this should now be apparent. If we were to insert a NIL into the end of the list, we get:

((new stuff) list ())



CDR of the above is:

(list ())



CDR of that is:

(())



CAR of that is:

() [NIL]

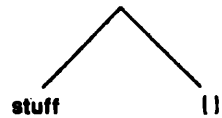
**NIL**

Both the CAR and CDR of NIL is NIL thus we can't change the list from here on without CONSing stuff onto it. Let's do so:

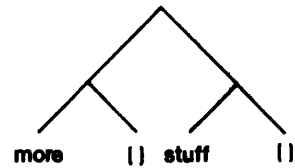
(setq worktree ())

**NIL**

```
(setq worktree (cons 'stuff worktree))
```



```
(setq worktree (cons '(more) worktree))
```



You should be able to draw and analyze the functions of CONC, and various other simple LISP functions.



My apple trees will never get across  
And eat the cones under the pines, I tell him.  
He only says, "Good fences make good neighbors."  
— Robert Frost

### Chapter 13: TREES AND RECURSION

Now, we promised you that trees would help you understand recursion. It really will. Most functions in LISP, whether they are built-in or we write them ourselves, are designed to work on trees. All trees in LISP [subtrees are trees also] look alike. Therefore, if we have a function that works on a tree it will work on any tree.

Our first version of RECITE took off the leftmost branch of the argument list [tree] with CAR and printed it. It then recursed [it called itself] passing the rest of the tree [the CDR] as the new tree. Unfortunately, if any of the branches were themselves trees then they would simply be printed out as-is:

```
:(define (recite (lambda (stuff)
:   (cond ( (null stuff) () )
:         ( t (print (car stuff))
:             (recite (cdr stuff)) )
:         ) )))
RECITE
:(recite '((we the people) star (e plenebla) trek))
(WE THE PEOPLE)
STAR
(E PLENEBLA)
TREK
NIL
```

How can we get around this? The thing that should come to mind is that RECITE will work on any tree. Thus, if before RECITing the CDR of the list we make sure that all the subtrees in the CAR of the list have been RECITED we should be home free. No matter how deeply nested the main tree is, we will eventually get to its leaves by calling RECITE over and over again on deeper and deeper subtrees until we hit one whose CAR is an atom.

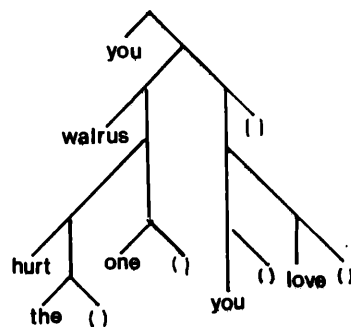
We will define RECITE with a COND as suggested by the previous description. We keep calling RECITE of the CAR [and then CDR] of the things that are entered until we hit an atom and then simply print out the atom! Would that give us the same result?

Watch!

```
:(define (recite (lambda (stuff)
:   (cond ( (atom stuff) (print stuff) )
:         ( t (recite (car stuff))
:             (recite (cdr stuff)) )
:         ) )))
RECITE
:(recite '(you (walrus (hurt the) one) ((you) love))))
YOU
WALRUS
```

HURT  
THE  
NIL  
ONE  
NIL  
YOU  
NIL  
LOVE  
NIL  
NIL  
NIL

What happened? There are extra NILs in the way. Let's look at the input tree:  
(you (walrus (hurt the) one) ((you) love))



When we finally get down to printing "the" ~~[The value of STUFF is indicated by the "\*" in the above tree]~~ the CDR of STUFF is a NIL which qualifies as an atom so it gets printed out also. Let's put in a test for that NIL and simply do nothing when we encounter it!

```

(define (recite (lambda (thing)
: (cond ((null thing) nil)
: ((atom thing) (print thing))
: (t (recite (car thing))
: (recite (cdr thing)))
: ))
RECITE
(recite '(you (walrus (hurt the) one) ((you) love))))
YOU
WALRUS
HURT
THE
ONE
YOU
LOVE
NIL

```

One of the important uses of recursion is tree searching. RECITE is an example of this method. By using the function recursively and passing the later calls smaller and smaller trees we can scan the entire main tree.

But no one style, I think, is recommended.  
 --Richard Purdy Wilbur

## Chapter 14: A STYLE OF PROGRAMMING

The result of a simple recursive function is that the value returned is the value of the last expression evaluated that does not involve a recursive call. Typically, this is a T or NIL or some atomic value. Since the function is stopped at the point of call when a recursion is invoked we can use the result of a call as the argument to some function. This section will deal with this type of more complex recursion.

Suspended evaluation is of primary importance here. When a function is evaluating its arguments it is suspended until all those arguments are done evaluating. [We've been over this many times, once more can't hurt.] So, if in the process of evaluating arguments a recursive function call is involved, that recursion takes place without disturbing the suspended evaluation even though the recursion likely involves another occurrence of that expression.

The only way to really understand what we're driving at is to see it happen. TRACE is the fastest way to do this and thus it should be used freely in your examination of this chapter.

The first example is a function to flatten a list. This function will return all of the atoms in a single list with no nested lists, that is, it will remove all the nesting parentheses. If we view a list as a tree, then this function will return a list of all of the "leaves". Note the similarity of this function to the RECITE function of the last few chapters. [We introduce the LIST function here. It is the built-in version of MAKELIST which we mentioned earlier.]

Here is the function definition:

```
(define (flatten (lambda (l)
:   (cond
:     ((null l) nil)
:     ((atom l) (list l))
:     (t (conc (flatten (car l))
:              (flatten (cdr l)))) ) )))
  FLATTEN
:(flatten '(a b c))
  (A B C)
:(flatten '(((your))) (((face))))
  (YOUR FACE)
:(flatten '(((tweedledum) (((and))) (tweedledee)))
  (TWEEDLEDUM AND TWEEDLEDEE)
```

The first condition in the COND phrase takes care of the case where the list is NIL. The next case handles an atom by making it into a list. The last case causes a recursion if the argument L is neither a null list or an atom. It flattens the CAR of L, flattens the CDR of L, and then CONC's the two results together.

There is a certain style to these recursive functions, and it is now the time to expand upon this. We always use the same format: first handle the termination conditions, then deal with special cases, then do the general case assuming that the special cases are handled properly. This formula is the way of LISP!

Some care should be taken in setting up the special cases. We test for NULL before we test for ATOM. If you do not see why this is the case, scrutinize this next segment of output.

```

(define (evil-flatten (lambda (l)
  (cond
    ((atom l) (list l))
    ((null l) nil)
    (t (conc (evil-flatten (car l))
              (evil-flatten (cdr l)))) ) )))
EVIL-FLATTEN
(evil-flatten '(eins zwei drei))
(EINS ZWEI DREI NIL)
(evil-flatten '((loops) (bloops) hoops))
(OOPS NIL BLOOPS NIL HOOPS NIL)

```

The reason, if you didn't figure it out, is that NIL is an atom so that testing for ATOM of NIL will be T and the LIST of NIL will be returned. This is undesirable, and underscores the necessity of setting up the termination conditions of the recursion correctly.

On to the next example. Here we are concerned with writing our own version of the REVERSE function. [Review the behavior of this function if you don't remember how it works.] The game plan has a little trick to it. But first, let's borrow the shell of the "standard" recursive function:

```

(define (rev (lambda (l)
  (cond
    ((null l) nil)
    (t (*****)) ) )))

```

where the asterisks indicate the general case of the function which we haven't written yet. How should we proceed?

The trick is: suppose that REV works! Then: (REV (CDR L)) is the reverse of the list without its CAR. If we stick the CAR back on the right end of this expression then we have the reverse function. Let's see it in full:

```

(define (rev (lambda (l)
  (cond
    ((null l) nil)
    (t (conc (rev (cdr l))
              (list (car l)))) ) )))
REV
(rev '(test a is this))
(THIS IS A TEST)
(rev '((phoo bar) (food bar) (poo bear)))
((POO BEAR) (FOOD BAR) (PHOO BAR))

```

Success!

You may have some qualms about our method of "assume it works". What kind of a crazy method is it that allows you to assume your functions work when you haven't yet finished writing the stupid thing?

It is a very useful and quite valid method. Consider the definition of the factorial function. [The factorial of  $n$  is the product of the first  $n$  positive integers.]

$$n! = \begin{cases} 1, & \text{if } n=0 \\ n*(n-1)!, & \text{if } n>0 \end{cases}$$

The definition of the factorial function exhibits the same behavior as our recursive functions: first set up the termination conditions [here,  $n=0$ ] and then define the general case in terms of similar, simpler uses of the same function [here,  $n!=n*(n-1)!$ .] It works for mathematicians and it works for LISP programmers!

Since we have gotten this far, let's define the factorial in LISP.

```
:(define (factorial (lambda (n)
:   (cond
:     ((equal n 0) 1)
:     (t (mult n (factorial (sub n 1)))) ) )))
  FACTORIAL
:(factorial 0)
  1
:(factorial 1)
  1
:(factorial 2)
  2
:(factorial 3)
  6
:(factorial 5)
  120
```

Here is one more recursive example: the function RAC rewritten with recursion. What is the strategy here? The recursion phrase is easy: keep taking the CAR of the list, removing the first element of the list, until we get to the end of the list. So far we have:

```
(define (rac (lambda (l)
  (cond
    (*****
    (t (rac (cdr l))) ) )))
```

The termination condition is a little slippery. If we make it: `((null l) nil)`, then the function will keep calling itself, dropping off the first elements, until it gets to a NIL list, and then return the NIL. This isn't quite what we had in mind. What we want to do is to stop recursing just before we get to the end of the list. Try this termination phrase: `((null (cdr l)) (car l))`. What this will do is stop the recursion one call before the list becomes NIL. That is, if the CDR of the list is NIL, then the first element of the list is the last element. The whole function becomes:

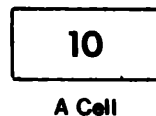
```
:(define (rac (lambda (alist)
:   (cond
:     ((null (cdr alist)) (car alist))
:     (t (rac (cdr alist))) ) )))
  RAC
```

```
:(rac '(a b c d))  
D  
:(rac '(bird (nest)))  
(NEST)
```

## Chapter 15: SCOPE CONSIDERATIONS

LISP functions live, work and play in an environment. The purpose of this chapter is to describe this environment and the effects of lambda-binding on it.

When you enter the LISP system, there are some predefined LISP functions [the built-ins] and two pre-defined variables, T and NIL. Using SETQ to define your own variables causes LISP to set up an area of storage to hold the value of that variable. Thus, evaluating the expression: (SETQ A 10), LISP binds the value of 10 to the variable A, setting up something like this:



Since we are not currently evaluating any function, this value of A is said to be in the "global" environment. The global environment is the state of affairs at the highest level [that is, not inside any functions]. Any subsequent use of the SETQ function with A will change the value in the A cell.

Of interest here is how the assignment interacts with the evaluation of user-defined functions. Remember that the global values of variables are left untouched even though the local variables in the function may conflict in name. Let's define some functions with which we can experiment.

```

(setq x 10)
10
(setq y 20)
20
(define (fun1 (lambda (x)
: (print x)
: (print y)
: (fun2 x y)
: (fun3 (add x 1)) )))
FUN1
(define (fun2 (lambda (x z)
: (print x)
: (print y)
: (print z)
: (setq y 5)
FUN2
(define (fun3 (lambda (r)
: (print r)
: (print x)
: (print y) )))
FUN3
(fun1 2)

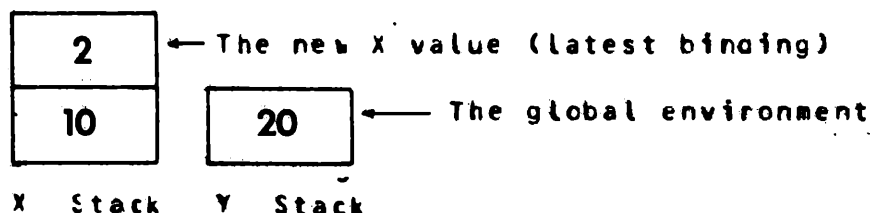
```

```

2
20
2
20
20
3
2
5
5
tx
  10
ly
  5

```

We defined three somewhat contrived functions. We start off the process of evaluation by typing "(fun1 2)". When FUN1 is evaluated, the value of its actual argument 2 is bound to the local variable X. But X already exists in the global environment by means of the SETQ function. LISP always checks to see if a conflict between a global variable and a local variable exists. If there is such a conflict the new value is "stacked" on top of the older value or values. This can be imagined with the help of the following picture:



The value of the atom in evaluation is given by the topmost value on its stack. Thus the value of X at this point is 2 and the value of Y is 20. The values on the top of the stack are called a local environment.

After the function in which the local variable is defined ends, the topmost [current] value is removed from the stack. Thus, the older value is returned. Note that the global environment can't be removed from the stack in this way.

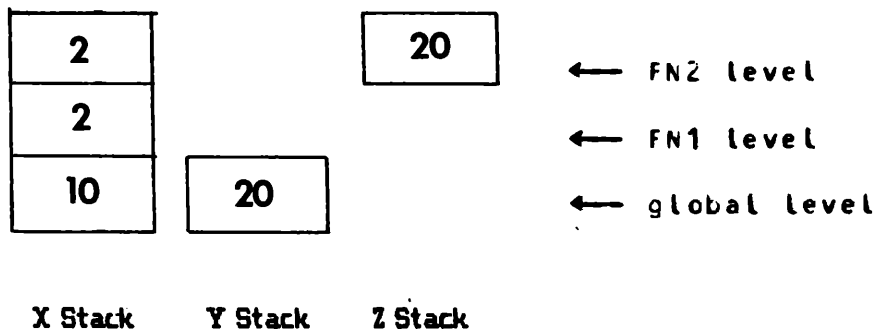
While inside the function, the local value acts like a global value to all the functions called by the first function. The same is true for all deeper levels of function calls.

Back to our example. We are now inside the function FUN1. The first expression to be evaluated is: "(PRINT X)". This causes the current value of X to be printed. Since X is a local variable to this function, the current value of X is the value of the argument supplied in the call of the function FUN1. This is the number 2 which we typed ourselves. Thus, the first two in the output.

The next expression: "(FUN2 X Y)" is a call to the function FUN2. The formal arguments for FUN2 are X and Z. Since X already exists in the environment [twice, in fact] we must again "stack" the new value, 2, on top of the older one, 2. [Note that the old value and the new value

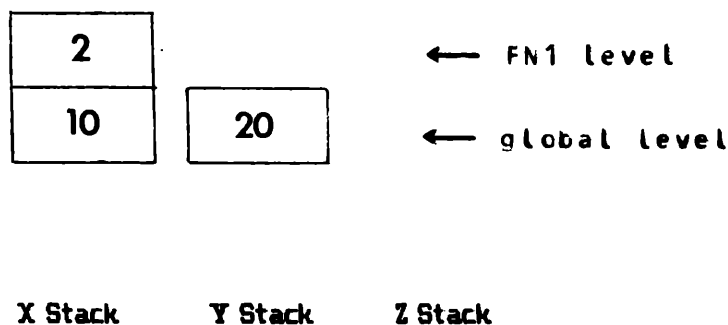


are the same. LISP does not care, it will save the old one anyhow.] Also, the value of 20 is bound to Z. The new environment looks like this:

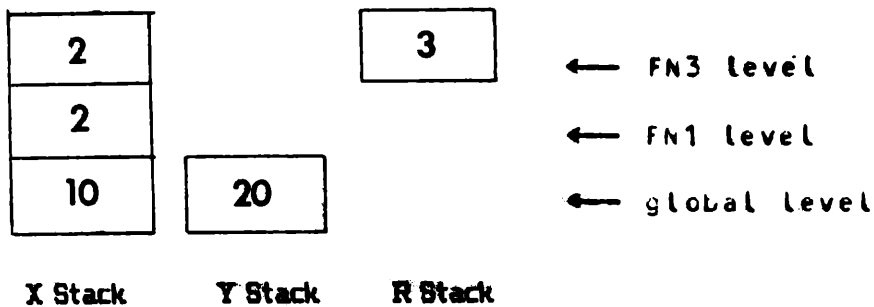


We are now inside of FUN2 [which is inside of FUN1]. The value of X is printed [2, the third output line]. Then, the value of Y is printed. This is a 20. Next, the value of Z is printed [20, the fifth line]. We then use the SETQ function to assign a value to the variable Y. Therefore, Y now has the value of 5. If Y had been a local variable in FUN2, then the assignment would have been broken upon leaving FUN2. However, since Y is global, this assignment of value is permanent, at least until the next assignment. The action of changing a value of a variable that is not one of the functions formal arguments like this is called a "side effect". Side effects are usually very nasty, and should be avoided.

FUN2 now exits, returning the value of Y [which is thrown away]. We are now back in FUN1 and the topmost level has been removed from the stack.



We then call FUN3 with the expression "(FUN3 (ADD X 1))". The value of X inside FUN1 is 2. Therefore (ADD X 1) evaluates to 3. The value of the formal argument in FUN3, namely R has the value of 3 bound to it.



The value of R is now printed on the sixth line. The variables X and Y are also printed. The question is, which X and which Y? Neither are local to FUN3. For X we use the value which was available in the previous environment, that of FUN1. There, X had the value 2. Therefore, a 2 is printed for X. What about Y? Well, Y isn't local to any function so LISP uses the global value of 5. [Note that the global value was assigned from inside the function FUN2.]

The end. The last number printed is the value returned from FUN1, which is returned from FUN3, which is returned from PRINT. Remember that everything in LISP returns a value, even PRINT.

Lisp! To speak imperfectly or like a child. . . .

## Chapter 16: A CATALOG OF LISP FUNCTIONS

This chapter represents a departure from our previous format. Here we present, without commentary, some common LISP functions. These are functions which tend to pop up a lot. They are here so that you can learn to read LISP functions and also to provide some examples of what "good" LISP functions look like. As you learn to read, the little syntactic details like where the parentheses go will become insignificant, allowing you to concentrate on the "logic" of a function, rather than its form.

We are not supplying explanations. It is your assignment to determine how each function works. To facilitate this, we are also giving examples of each function's use.

```
:(define (maxlist (lambda (l)
:   (cond
:     (null l) nil)
:     (null (cdr l)) (car l))
:   ((greater (car l) (car (cdr l))) (maxlist (cons (car l) (cdr (cdr l)))))
:   (t (maxlist (cdr l)))))
```

MAXLIST

```
:(maxlist '(1 4 4 7))
```

7

```
:(maxlist '(10 1 0))
```

10

```
:(maxlist '(1))
```

1

```
:(define (exp (lambda (n m)
:   (cond
:     ((zero m) 1)
:     ((equal n 1) n)
:     (t (mult n (exp n (sub m 1)))) )))
```

EXP

```
:(exp 3 2)
```

9

```
:(exp 2 3)
```

8

```
:(exp 3 3)
```

27

```
:(define (index (lambda (a l)
:   (cond
:     ((null l) nil)
:     ((null (member a l)) 0)
:     ((equal a (car l)) 1)
:     (t (add (index a (cdr l)) 1)))))
```

## INDEX

```
:(index 'a '(a b c d))
```

```
1
```

```
:(index 'index '(maxlist ascending index exp))
```

```
3
```

```
:(index '(help me) '(please (help me) said the man))
```

```
2
```

```
:(define (snoc (lambda (l a)
```

```
:   (cond
```

```
:     ((null l) (list a))
```

```
:     (t (cons (car l) (snoc (cdr l) a))) )))
```

```
  SNOC
```

```
:(snoc '(this is a) 'test)
```

```
  (THIS IS A TEST)
```

```
:(snoc '(car rac cons) 'snoc)
```

```
  (CAR RAC CONS SNOC)
```

```
:(snoc '((alice in)) '(wonderland))
```

```
  ((ALICE IN) (WONDERLAND))
```

```
:(com please note that this is the same as the built-in APPEND)
```

```
:(define (assoc (lambda (a l)
```

```
:   (cond ((null l) ()))
```

```
:   ((equal (car (car l)) a) (car (cdr (car l))))
```

```
:   (t (assoc a (cdr l))))))
```

```
  ASSOC
```

```
:(setq alist '((x 10) (y 20) (z (a little lambda))))
```

```
  ((X 10) (Y 20) (Z (A LITTLE LAMBDA)))
```

```
:(assoc 'x alist)
```

```
  10
```

```
:(assoc 'z alist)
```

```
  (A LITTLE LAMBDA)
```

```
:(assoc 'w alist)
```

```
  NIL
```

```
:(define (replace (lambda (x y l)
```

```
:   (cond ((null l) ()))
```

```
:   ((atom l)
```

```
:     (cond ((equal l x) y)
```

```
:     (t l)))
```

```
:   (t (cons (replace x y (car l))
```

```
:     (replace x y (cdr l))))))
```

```
  REPLACE
```

```
:(replace 'always 'nevermore '(quoth the raven always))
```

```
  (QUOTH THE RAVEN NEVERMORE)
```

```
:(replace 'am 'was '(((i) (am that) ((i)) am)))
```

**((((I) (WAS THAT) (I))) WAS))**



Journey all over the universe in a map, without the expense  
and fatigue of travelling, without suffering the inconveniences  
of heat, cold, hunger, and thirst.  
---Cervantes (Don Quixote)

### Chapter 17: MAPS

We are now going to make life much simpler, at least with respect to LISP. The careful reader may have noticed some patterns in the recursive LISP functions we have been defining. Such is, in fact, the case. There are some patterns and LISP provides some functions to perform a number of these.

Suppose that we want to perform an operation on each element of a list, in succession. There is a LISP function to do this: MAPCAR. Let's first try an example.

```
(mapcar 'print '(all the news that fits))
ALL
THE
NEWS
THAT
FITS
(ALL THE NEWS THAT FITS)
```

What happened? Well, we used the PRINT on each element of the list. This caused the first five lines of the result. [Each PRINT took up one line.] LISP then collected all of the results of the PRINTs into a list and returned this collection as the result of the expression. We normally won't want to use the PRINT function on each element of a list; let's try some more interesting examples.

```
(mapcar 'even '(0 1 2 3 4 5))
(T NIL T NIL T NIL)
(mapcar 'length '((a) (b b) (c d e)))
(0 1 2 3)
(mapcar 'reverse '((are we) (in not) (toto kansas)))
((WE ARE) (NOT IN) (KANSAS TOTO))
```

The exact definition for MAPCAR is as follows: MAPCAR takes as its arguments a function name and a list. First, use the given function on the CAR of the list. Then continue using the same function with the CDR of the list. When finished, make a list of all the individual results.

In the first example above, LISP applied the function EVEN to each element of the list: "(0 1 2 3 4 5)". In the second example, LENGTH was used on each element of its associated list. In the last example, we used the REVERSE function on each element. Note that this is sort of the complement of the regular use of the REVERSE function which REVERSEs all of the top level elements of a list. Here we REVERSEd each element, but we left the top level untouched.

A recursive LISP function equivalent to the first example above is presented below.

```
(define (evenmap (lambda (l)
: (cond
: ((null l) nil)
```

```
:      (t (cons (even (car 1))
:              (evenmap (cdr 1)))) ))))
EVENMAP
:(evenmap '(0 1 2 3 4 5))
(T NIL T NIL T NIL)
```



## Chapter 18: ISPLAY OGRAMMINGPRAY

You now have all the tools needed to solve some reasonably large problems in LISP. [You've actually had most of them for a while.] This chapter demonstrates the steps that can be taken to solve such problems. The particular steps are, of course, dependent upon the exact problem specified but there are some general rules which we will emphasize as they go by.

The problems we will attack is that of a pig Latin translator [to, not from]. The system [by system, we mean a collection of functions and help functions all intended to solve one problem] will take a sentence in English and return the sentence in pig Latin. The system will do this!

```
!(piglatin '(take out the trash))
(AKETAY OUTWAY ETHAY ASHTRAY)
```

The first step in designing a system is!

Define the problem and lay out exact specifications.

In this case, the specifications are pretty easy! The user will enter his sentence as a list of words and the program will return the sentence with each word "pigized".

Since you need to know what you're doing before you start writing the program, the next step is!

Decide on an algorithm.

The standard pig Latin algorithm goes as follows!

Look at each word in the sentence. For each word, if its first letter is a vowel, then simply add "way" to it. Otherwise, remove all the letters up to the first vowel from the front of the word and put them after the word, then add "ay".

There are a few assumptions in this description. It is very very important to specify your assumptions so that you know which cases you don't have to deal with. A well-defined set of assumptions can make a programming task quite a bit simpler.

We have assumed, first of all, that all words have a vowel someplace in them. We will ignore the case of having words with no vowels in them. We assume, somewhat implied by our first assumption, that only vowels can be one-letter words [like "a" or "I"]. What are our vowels? By examining some sample cases, we can fill in blanks in the algorithm.

```
WATER --> ATERWAY
WEEPS --> EEPWAY
```

Thus, "W" is not a vowel.

```
YELLOW --> YELLOWAY
YAKS --> YAKSWAY
```

So "Y" is a vowel. Also, obviously, "A", "E", "I", "O", and "U" are vowels.

We've now pretty completely specified the problem. How do we go about programming it? The most useful technique to learn is called "top down programming". It means that the main programs are written before the help functions. By using the top down style we can organize our thoughts.

Let's start at the very beginning [a very good place to start]. The first part of the algorithm says: "Look at each word in the sentence." Since our sentence is a list, the words are the atoms in that list. Looking at each word is like the MAPCAR operator [we told you it was commonplace]. Thus, let's define our main function to use MAPCAR and cause each word in the sentence to be processed.

```
(define (piglatin (lambda (sentence)
:      (mapcar 'pigword sentence) )))
PIGLATIN
```

This is the main function. There are two important factors to note in the definition of this function. First, its name [and the name of the help function it calls] are meaningful. We could just as easily called this function "xyzyz" but then we would never be able to remember what it did!

The second point is a little more subtle. The functions, the main one and all the functions that we'll write are very short. We took a single idea [mapping down the sentence] and turned it into a function. The help function ["pigword"] will also implement one little part of the whole. By slowly adding parts of the algorithm we can build the entire system in very small, easily managable increments. There are a lot of advantages to this, not the least of which is that simpler ones are easier to edit.

Let's get on to the first help function! the processing of each word. The PIGWORD function will take a word and turn it into pig Latin. The I/O [standing for Input/Output] behavior of this function should be:

```
(pigword 'this)
ISTHAY
```

There's a problem here. We have functions that modify lists but not functions for modifying the letters in an atom. The solution is to make the atom into a list and then turn the processed list back into an atom like this:

```
THIS -> (T H I S) -> (I S T H A Y) -> ISTHAY
```

Of course, LISP provides functions for doing this conveniently. EXPLODE turns an atom into a list of its letters and IMplode does the opposite [we shouldn't have to give you examples of everything by this time].

Here we have a new concept not specified in the algorithm but implicit in the nature of LISP. We need to have some intermediate processing step that does this explosion, and subsequent impllosion. We will define PIGWORD as follows:

```
(define (pigword (lambda (word)
:      (implode (piglistedword (explode word)))) ))
```

**FIGWORD**

**FIGLATIN** calls **FIGWORD** via **MAPCAR** and passes it each word. **FIGWORD** in turn explodes the word and processes it using another help function [with a meaningful name] called **FIGLISTEDWORD**. **FIGLISTEDWORD** will return the list of the "pigized" word and **FIGWORD** will implode it to an atom again and return the new word to **FIGLATIN**. End of Operation!

All that remains to do now is write **FIGLISTEDWORD**. Which part of the problem does this one implement? Here is the definition:

```
(define (piglistedword (lambda (word)
  (cond ( (isavowel (car word)) (pigvowel word))
        (t (pignovowel word) )
        ) )))
FIGLISTEDWORD
```

Several things are still missing from the system, **PIGVOWEL** will translate a word, represented in list form, into its pig Latin equivalent, **PIGNOVOWEL** will do all other words, and the predicate **ISAVOWEL** will tell us whether the letter passed to it is a vowel or not.

Let's write the easy ones first!

```
(define (isavowel (lambda (letter)
  (member letter '(a e i o u y)) )))
ISAVOWEL
(define (pigvowel (lambda (word)
  (conc word '(w a y)) )))
PIGVOWEL
(pigvowel '(a f t e r))
(A F T E R W A Y)
```

We should now be able to test at least the easy part of the system — sentences where all the words begin with a vowel!

```
(piglatin '(alert airmen always are ontime))
(ALERTWAY AIRMENWAY ALWAYSWAY AREWAY ONTIMWAY)
```

[What would happen if we tried to "pigize" a sentence with words that began with consonants? Try it.]

Great -- most of our system is finished, and in only 5 functions. All we have left to do is write the hard one! **PIGNOVOWEL**. Let's describe the responsibility of that function in detail.

Since we can only really deal with one letter of the exploded word at a time, we have to look down the word until we hit a vowel [**ISAVOWEL** will be useful here]. When we find the vowel we will attach the first letters [which we will have been collecting up along the way.] to the end of the word and tack on an "ay".

How can we recur down a list and keep stuff as we go? The answer is that we pass the list of collected letters along with the recursion and just tack on letters along the way. This is the function, follow it by hand, see what it does!

```

(define (pignovowel (lambda (word lets)
: (cond ( (isavowel (car word))
:       (conc word (conc lets '(a y))) )
:       (t (pignovowel (cdr word)
:                       (append lets (car word))) )
: ) )))
FIGNOVOWEL

```

Note that there are two parameters to this function. We will always have to supply both of them or this function will not work. O.K., let's try it out!

```

:(pignovowel '(t h i s) ())
(I S T H A Y)

```

That looks good, we should now be able to try out the whole system!

```

:(piglatin '(every good boy does fine))
** ERROR: TOO FEW ARGS **
FIGNOVOWEL :: (WORD)
+()
NIL

```

Ooops! Something went wrong. The error message said we called FIGNOVOWEL with one argument instead of two. The call was from FIGLISTEDWORD. Oh yes, we hadn't foreseen that we'd need two arguments in FIGNOVOWEL when we wrote FIGLISTEDWORD. Oh well, we can now go and edit it or retype it.

```

(define (piglistedword (lambda (word)
: (cond ((isavowel (car word)) (pigvowel word))
:       (t (pignovowel word nil))
: ) )))
FIGLISTEDWORD
:(piglatin '(sticks and stones are painful))
(ICKSTAY ANDWAY ONESSTAY AREWAY AINFULPAY)

```

Looks good!

We've now completed the design of our pig Latin system to specification. You should SAVE it so that you can recover it later.

The lessons of this chapter are summarized in the following list:

- Formulate a complete specification of the problem before even beginning to think in LISP. The I/O behavior of the function should be the main part of this description. I/O behavior is a good way to formulate the descriptions of most help functions also.
- Define the algorithm and assumptions [restrictions] involved. This will save you from embarrassing trouble later on when you suddenly remember something that you forgot.
- Use top down programming. That is, start with the function that the user will type in [the user now is someone who will be using your programs rather than you]. Fill in the blanks as you get to them getting deeper and deeper into help functions.

- As you write parts of the system test the help functions thoroughly. This will save you debugging time in the long run.
- Use mnemonic names. The names of all functions, help functions and variables should have meaning to you and to others. [CAR and CDR are good examples of what not to use as names.]
- Keep your functions short — need we say more about this?
- Implement a single part of the algorithm in each function. This will also help keep them short. If the algorithm is sufficiently well described then there should be about one function per clause in the description.
- Make sure that the parts of the system are internally consistent [this is what went wrong with the arguments of PIGNOVOWEL].

The design of a system in LISP or any other programming language is like designing anything else. You take it step by step and fill in the unknowns when you come across them. Always have the final goal in mind.



Life is the totality of those functions which resist death.  
--Marie Francois Xavier Bichat

### Chapter 19: FEXPRs -- UNEVALUATING FUNCTIONS

Most of the functions that we've used thus far are classified as "EXPRs" [short for EXPRession]. An EXPR is a function that evaluates its arguments when it is called. ADD, TIMES, CAR, CONS, and many others are all EXPRs.

It is often useful to be able to write a function whose arguments will not be evaluated but rather passed "asis" to the function. A function that does not evaluate its arguments is called a "FEXPR".

The first flexibility that this affords us is that we can type things in without quotes. The Pig Latin example could have been simplified with a FEXPR in that we could have typed:

(PIGLATIN THIS IS A TEST SENTENCE)

instead of:

(PIGLATIN '(THIS IS A TEST SENTENCE))

A FEXPR has a slightly different DEFINE syntax than an EXPR. The word LAMBDA is replaced by the word FLAMBDA and there is exactly one formal argument:

```
:(define (printme (flambda (input)
:   input )))
PRINTME
```

We have defined a trivial function to simply return the value that gets bound to its formal argument. This will enable us to see what a FEXPR does with the arguments:

```
:(printme i have but one list to give to my country)
(I HAVE BUT ONE LIST TO GIVE TO MY COUNTRY)
```

The FEXPR simply makes a list of the unevaluated arguments and binds this to the single formal argument.

```
:(printme (car (bus truck cab)) (cdr (caddr cddar)) )
((CAR (BUS TRUCK CAB)) (CDR (CADDR CDDAR)))
```

The CAR and CDR functions above are not evaluated. As far as the FEXPR is concerned they are simply names exactly like "bus", "truck", "caddr", etc.

The above PRINTME function acts exactly like the QUOTE function that we have encountered all through our LISP studies. In fact, this is exactly what the "'" sign does. When we put a quote before a list or an atom it is interpreted as if we had typed "(quote ...)". QUOTE is a FEXPR that returns the name of its argument unevaluated.

```
:'(car (dont evaluate this))
(CAR (DONT EVALUATE THIS))
```

```
:(printme (car (dont evaluate this)))
  (CAR (DONT EVALUATE THIS))
:(quote (car (dont evaluate this)))
  (CAR (DONT EVALUATE THIS))
```

A good rule of thumb is that a FEXPR should always call an EXPR to do the work. It can typically do this by using MAPCAR to scan down the list of input elements. Using this rule we can rewrite PIGLATIN as:

```
:(define (piglatin (lambda (sent)
:   (mapcar 'pigword sent) )))
  PIGLATIN
```

You should try this and verify for yourself that it works as expected. You should be able to type in the sentence to be translated without parentheses or the quote.

The other advantage that FEXPR gives us is the ability to write functions that take an unspecified number of arguments. For example, we might want to write a function that takes a list of pairs of names and phone numbers and returns each pair in list form.

```
:(define (pairwise (lambda (in)
:   (segment in) )))
  PAIRWISE
:(define (segment (lambda (l)
:   (cond ((null l) ()))
:         (t (cons (list (car l) (cadr l))
:                   (segment (cddr l)) ))
:   )))
  SEGMENT
:(pairwise beth 555-1385 anne 202-4926 andy 901-6583)
  ((BETH 555-1385) (ANNE 202-4926) (ANDY 901-6583))
```

There is another important rule of FEXPRs! Never recur with a FEXPR and avoid calling them from within other functions. Why is that? Consider what the result of calling a FEXPR recursively will do. Let's define a recursive FEXPR:

```
:(define (revlist (lambda (l)
:   (cond ((null l) ()))
:         (t (cons (reverse (car l))
:                   (revlist (cdr l)) ))
:   )))
  REVLIST
```

This should reverse each element of the input list. Thus, we should be able to type:

```
(revlist (him rebuild) (was he than better))
```

and get

```
((rebuild him) (better than he was))
```



in response.

Watch what happens:

```
:(revlist (him rebuild) (was he than better))  
-->> REVLIST :: ((HIM REBUILD) (WAS HE THAN BETTER))  
-->> REVLIST :: ((CDR L))  
-->> REVLIST :: ((CDR L))  
-->> REVLIST :: ((CDR L))  
+()
```

Close, but no ananab! The first list went in okay, but it looks like the recursive steps didn't work right. Since the FEXPR doesn't evaluate its arguments, the "(cdr l)" wasn't evaluated so the next iteration simply tried to do REVLIST on the list "(cdr l)" as opposed to the CDR of "l". This would have gone on forever if we hadn't interrupted it.



## Chapter 20: EVAL and APPLY

In the space of this brief chapter, we will show you the entire LISP interpreter. Well, not really, but we will show you some functions which form the "heart" of the LISP system.

When you type an expression into the LISP system, it is passed to a function called EVAL. EVAL [short for "EVALuate"] processes your expression and returns the result. What is this thing called EVAL?

```

(setq a '(car b))
      (CAR B)
(setq b '(aa bb cc dd))
      (AA BB CC DD)
(eval a)
      AA
(car '(aa bb cc dd))
      AA
(eval 'a)
      (CAR B)

```

Let's see what happened. First we defined two variables, A and B. Note that the value of A is a legal LISP expression. When LISP EVALuates A, it is as if we had typed in the expression ourselves. LISP returns the value of the expression as the result. When LISP EVALuates "A", it returns the value of the value of 'A, which is the same thing as the value of A.

The importance of this ability may not be immediately apparent. However, notice that this enables us to manipulate programs as data and then evaluate them. Most other programming languages do not provide this facility. Here is a small example:

```

(setq fun 'mult)
      MULT
(setq x 3)
      3
(setq y 2)
      2
(setq vars '(x y))
      (X Y)
(eval (cons fun vars))
      6

```

In LISP, there is another function which will evaluate a function and its data: APPLY. The generic form of the APPLY function is: (APPLY function-name list-of-arguments). Thus, to repeat the last line in the above example using the APPLY function:

```

(apply fun vars)
      6

```

Let's apply what we know about EVAL to the problem of evaluating polynomials. The polynomials are going to be represented by their associated LISP expressions. Thus,

$$3x^2 + 15$$

will be represented as

```
(ADD (MULT 3 (EXP X 2)) 15)
```

Suppose we have this representation as the value of some variable.

```
:(setq p '(add (mult 3 (exp x 2)) 15))
(ADD (MULT 3 (EXP X 2)) 15)
:(eval p)
** ERROR: UNDEFINED ATOM **
EVAL :: X
+()
NIL
:(setq x 3)
3
:(eval p)
42
```

Now we have the capability to form polynomials and then evaluate them.

Nor shall any person...be deprived of life, liberty, or property without due process of law.  
 -- Fifth Amendment to the Constitution

## Chapter 21: PROPERTIES AND LAMBDA EXPRESSIONS

We have seen several ways to attach "meanings" to names [atoms]. The SETQ function gives a value to an atom. There is one other way of connecting values to atoms.

A property is a name associated with a particular value of an atom. As an analogy, think of an atom as a chest of drawers; the top drawer would contain something, the second would contain something different, etc. All the things in the drawers, however, are still associated with the chest [the atom].

Let's construct our chest of drawers:

```
(put 'chest 'top '(sox))
(SOX)
(put 'chest 'second '(underwear (shorts shirts)))
(UNDERWEAR (SHORTS SHIRTS))
(put 'chest 'third '(t-shirts jeans))
(T-SHIRTS JEANS)
(put 'chest 'bottom '(pajamas))
(PAJAMAS)
```

The PUT function takes three arguments. The first is the name of the atom we are attaching properties to ["chest"], the next is the name of the property ["top", "bottom", etc.], and the third is the value to attach to that atom at that property. This value can be anything at all [lists, names, numbers]. The GET function looks at properties on an atom:

```
(get 'chest 'second)
(UNDERWEAR (SHORTS SHIRTS))
(get 'chest 'top)
(SOX)
(setq place 'chest)
CHEST
(put place 'top
 (cons (get place 'top) (get place 'second)))
((SOX) UNDERWEAR (SHORTS SHIRTS))
(get place 'top)
((SOX) UNDERWEAR (SHORTS SHIRTS))
(rem place 'second)
NIL
(get place 'second)
NIL
```

In case you hadn't figured it out, REM removes a property from the property list. It's kind of like pulling out a drawer. We can't GET the value of that property after it has been REMed.

We said way back when that you couldn't take the CDR of an atom. That isn't quite true. The CDR of a name [a quoted atom] returns all the properties associated with that atom in the

form:

```
(property value property value property value ...)
```

```
:(cdr place)
  (BOTTOM (PAJAMAS) THIRD (T-SHIRTS JEANS)
   TOP ((SOX) UNDERWEAR (SHORTS SHIRTS)))
:(cdr 'chest)
  (BOTTOM (PAJAMAS) THIRD (T-SHIRTS JEANS)
   TOP ((SOX) UNDERWEAR (SHORTS SHIRTS)))
```

The value set by SETQ and the properties associated with the name are completely separate.

```
:(setq chest 5)
  5
:(cdr 'chest)
  (BOTTOM (PAJAMAS) THIRD (T-SHIRTS JEANS)
   TOP ((SOX) UNDERWEAR (SHORTS SHIRTS)))
:chest
  5
```

What are properties good for? Why are they in LISP?

For a simple example we might arrange our phonebook based upon our friend's name. Each name has associated with it a property "number" and a property "address". This isn't really a whole lot different than just having the names, numbers and addresses arranged as a list of triplets. The advantage of using the properties is that the process of finding someone's phone number or address is simply a matter of getting the right property from the atom which is the person's name:

```
:(put 'mary 'address '(123 front road))
  (123 FRONT ROAD)
:(put 'mary 'phone '(345 6789))
  (345 6789)
:(cdr 'mary)
  (PHONE (345 6789) ADDRESS (123 FRONT ROAD))
:(put 'cave 'address '(321 tronf street))
  (321 TRONF STREET)
:(put 'cave 'phone '(we7 1212))
  (WE7 1212)
:(get 'cave 'address)
  (321 TRONF STREET)
```

This is pretty useless because we are restricted to using address parts and phone numbers that are LISP atoms and, anyway, we could have done the whole thing with recursion and gotten the same result. However, as an exercise it can't hurt.

Another possible use of properties is to "tag" names. For example, let's imagine that we were going to type in a dictionary and wanted to tag each word that we typed with its part of speech. We also might want to include some other identifications like number [for nouns] or transitivity [for verbs]. By using PUT and GET to attach properties to the atom whose name is

the word we can accomplish this tagging quite simply:

```
(put 'aardvark 'pos 'noun)
  NOUN
(put 'aardvark 'number 'singular)
  SINGULAR
(put 'eat 'verbttype 'transitive)
  TRANSITIVE
(put 'soups 'pos 'noun)
  NOUN
(put 'soups 'number 'plural)
  PLURAL
```

If we want to retrieve all of the parts of speech [pos] from a list of words, we could use MAPCAR with a function which will return the pos-property from a word. Here is the function PARTS which does just that.

```
(define (parts (lambda (sentence)
:      (mapcar '(lambda (word) (get word 'pos)) sentence))
:      )))
  PARTS
(parts eat aardcark soups)
  (VERB NOUN NOUN)
```

What, you might well ask, was all that about?!?!? Looks like we half-wrote a function in the middle of another one! The expression "(LAMBDA (WORD)... 'POS)" is typical of what we type for the definition of a function using DEFINE.

Let's look at some simpler examples:

```
(setq fn '(lambda (x) (reverse x)) )
  (LAMBDA (X) (REVERSE X))
(fn '(the value of fn is a lambda))
  (LAMBDA A IS FN OF VALUE THE)
(' (lambda (x) (reverse x)) '(this one is right here))
  (HERE RIGHT IS ONE THIS)
```

LAMBDA expressions, variables whose values are LAMBDA expressions, or expressions which evaluate to LAMBDA expressions can be used in a LISP expression in any place a function name would normally occur. A LAMBDA expression is sort of a temporary function. The appropriate values of its arguments are bound during evaluation, but after the result is returned, the function, and the argument values, go away.

When we use DEFINE to establish a function definition, it puts the LAMBDA expression forming the body of the function as a property of the function name. The property where this function is stored is called EXPR.

```
(cdr 'parts)
  (EXPR (FLAMBDA (SENTENCE) (MAPCAR (QUOTE (LAMBDA
  (WORD) (GET WORD (QUOTE POS)))))))
(get 'parts 'expr)
  (FLAMBDA (SENTENCE) (MAPCAR (QUOTE (LAMBDA
```

**(WORD) (GET WORD (QUOTE POS))))))**

In general, LISP looks at the world as follows:

- 1) Everything is an expression [i.e., has a CAR and a CDR or is an atom].
- 2) If the expression is an atom then return its value.
- 3) If the expression is a list then apply rule 4 to the CAR and use the CDR as the arguments to the function referred to in rule 4.
- 4) If the CAR is an atom then either its value is a LAMBDA expression [as example 3 above] or it has an EXPR on its property list whose value [i.e., (GET(CAR expression) 'EXPR)] is a LAMBDA expression [as in all the functions created by DEFINE]. Evaluate that LAMBDA expression!
- 5) If the CAR is a list, evaluate it and go to rule 4.

That's all a bit arabesque. Perhaps a few examples would help out. First, let's suppose that the variable [atom] X has the value: "(lambda (f) (reverse f))".

We enter:

**((CAR '(X Y Z)) '(LIST TO BE REVERSED))**

X [the result of CAR...] evals to the form:

**((LAMBDA (F) (REVERSE F)) '(LIST TO BE REVERSED))**

The F binds to the argument. The new expression is:

**(REVERSE '(LIST TO BE REVERSED))**

which returns:

**(REVERSED TO BE LIST)**

We could have equivalently used DEFINE to jam the LAMBDA expression into the EXPR property of the atom X. The evaluation would have worked in the same way.

In a previous chapter we asked about whether define was an EXPR or a FEXPR. Since we now know what DEFINE really does to things we can define it. This seems a bit redundant, and it is, but it is a good exercise.

DEFINE is of the form:

**(DEFINE (name (LAMBDA-expression))**

Since (name (LAMBDA-expression)) can't be evaluated [especially before the name is defined] we have to use a FEXPR in order to keep LISP from trying to evaluate that. Thus our first line must be:

**(DEFINE (DEFINE (FLAMBDA (function-form)**

The function-form will have the form:

**(name (LAMBDA-expression))**



Now, our task is easy. Let's redefine **DEFINE** in real LISP and see if it works as expected. If you try and do this, it might be a good idea to call it something other than **DEFINE**, since if you make a mistake, you can be in severe trouble.

```

:(define (define (flambda (functionform)
:  (put (caar functionform) 'expr (cadar functionform))
:  )))
  DEFINE
:(define (endof (lambda (s)
:  (car (reverse s))
:  )))
  (LAMBDA (S) (CAR (REVERSE S)))
:(cdr 'endof)
  (EXPR (LAMBDA (S) (CAR (REVERSE S))))
:(endof '(a s d f))
  F
:(cdr 'define)
  (SUBR * EXPR (FLAMBDA (FUNCTIONFORM) (PUT (CAAR FUNCTIONFORM)
  (QUOTE EXPR) (CADAR FUNCTIONFORM))))
:(rem 'define 'expr)
  NIL
:(cdr 'define)
  (SUBR *)

```

Note that after we redefined **DEFINE** we are using only the new value of **DEFINE** [ours]. The property that you see in the last line above [**SUBR**] holds the real value of **DEFINE**. When we **REM** our **EXPR** definition from **DEFINE**'s property list the old value [**SUBR**] comes back [DON'T FORGET TO DO THIS]! Don't worry about what a **SUBR** really is, we will discuss that on the chapter about internals.

LISP functions exist as properties of atoms with the name of the atom being the name of the function. Since LISP functions are only LISP expressions, you can see how being able to manipulate these expressions can be useful. For one thing, it means we can write our own editor in LISP. It also means that we can write functions which generate other functions during their evaluation.



Anyone who cannot cope with mathematics is not fully human. At best he is a tolerable subhuman who has learned to wear shoes, bathe, and not make messes in the house.

--Robert Heinlein, The Notebooks of Lazarus Long  
in Time Enough For Love

## Chapter 22: DIFFERENTIATING POLYNOMIALS

We are now going to travel back in time to the days of freshman calculus. We are going to write a system which will perform symbolic differentiation of polynomials. This may sound impressive, but in LISP it is quite simple.

Here are the rules for differentiation which we will use:

$$D[0,x]=0$$

$$D[x,x]=1$$

$$D[(u+v),x]=D[u,x]+D[v,x]$$

$$D[(u-v),x]=D[u,x]-D[v,x]$$

$$D[(uv),x]=uD[v,x]+vD[u,x]$$

$$D[(u^n),x]=nu^{n-1}D[u,x]$$

We are using the capital letter "D" to indicate the differentiation operator. Also, we specify the variable with which the differentiation is being done with respect to.

Notice that some of these rules are recursive, e.g., in order to differentiate  $(u+v)$  we need to differentiate  $u$  and  $v$ .

So much for the specification of the problem. Let's recall the representation of polynomials in LISP from previous chapters, where polynomials were transformed into LISP lists. Thus, " $2x$ " translates to " $(MULT\ 2\ X)$ ", etc. We are going to write some help functions which return the different parts of the polynomial for use in our derivative function. We will need the outermost [or highest level] function in a polynomial. This is the left-most function in the LISP list. We will also be wanting the first and second term in the polynomial. Here is a picture.

```
(ADD (MULT 2 X) 3)
-----
|      |      |
|      |      | second term
|      | first term
| top-level function
```

Here are some of our help functions. The "function" in a polynomial is the CAR of the polynomial represented in LISP. The first and second terms of a polynomial are the CADR and

the CADDR of the LISP representations, respectively. Therefore:

```

(define (function (lambda (poly)
  :   (car poly) )))
  FUNCTION
(define (firstterm (lambda (poly)
  :   (cadr poly) )))
  FIRSTTERM
(define (secondterm (lambda (poly)
  :   (caddr poly) )))
  SECONDTERM
(setq p '(add (sub x 2) 12))
(ADD (SUB X 2) 12)
(function p)
ADD
(firstterm p)
(SUB X 2)
(secondterm p)
12

```

Note that none of these functions are strictly necessary. However, if we were to change the underlying LISP representation then it would only be necessary to change these three functions. If we didn't use them, then any change in the representation would require changing every access of the representation in all of the functions we write.

Let's write the main function first. It will be called DERV and take two arguments: a polynomial and the variable with which the polynomial is to be differentiated. Here is some sample behavior:

```

(derv '(add x 2) 'x)
(ADD 1 0)
(derv '(mult x 2) 'x)
(ADD (MULT X 0) (MULT 2 1))
(derv '(exp x 2) 'x)
(MULT (MULT 2 (EXP X 1)) 1)

```

We now exhibit the function DERV.

```

(define (derv (lambda (poly var)
  :   (cond
  :     ((atom poly) (dervatom poly var))
  :     ((equal 'add (function poly))
  :       (dervsum poly var))
  :     ((equal 'sub (function poly))
  :       (dervminus poly var))
  :     ((equal 'mult (function poly))
  :       (dervprod poly var))
  :     ((equal 'exp (function poly))
  :       (dervexp poly var))
  :   )))
  DERV

```

If the polynomial is an atom then we call a help function, DERVATOM, which will properly differentiate an atom. We will write DERVATOM shortly. The next four conditions compare the main function in the polynomial with the different functions we are using: ADD, SUB, MULT, and EXP. If one of those four are found, the appropriate help function is called.

Let's write DERVATOM. The derivative of an atom is equal to 1 if the atom is the variable with which the differentiation is being performed, and a 0 in all other cases. This function is quite simple to write.

```

(define (dervatom (lambda (poly var)
  (cond
    ((equal poly var) 1)
    (t 0 ))))
  DERVATOM
  (derv '1 'x)
    0
  (derv 'x 'x)
    1

```

We will now write the function for differentiating a sum of two polynomials. Here it is.

```

(define (dervsum (lambda (poly var)
  (list 'add
    (derv (firstterm poly) var)
    (derv (secondterm poly) var) )))
  DERVSUM
  (dervsum '(add x 3) 'x)
    (ADD 1 0)

```

Notice that DERVSUM, which is called by DERV, also calls DERV. Thus we have a PAIR of recursive functions.

Similarly, here is the function for differentiating a difference of two polynomials.

```

(define (dervminus (lambda (poly var)
  (list 'sub
    (derv (firstterm poly) var)
    (derv (secondterm poly) var) )))
  DERVMINUS

```

The functions for multiplication and exponentiation are only slightly more difficult.

```

(define (dervprod (lambda (poly var)
  (list 'add
    (list 'mult
      (firstterm poly)
      (derv (secondterm poly) var) )
    (list 'mult
      (secondterm poly)
      (derv (firstterm poly) var) )) )))
  DERVPROD
  (define (dervexp (lambda (poly var)

```

```

:      (list 'mult
:          (list 'mult
:              (secondterm poly)
:              (list 'exp
:                  (firstterm poly)
:                  (sub (secondterm poly) 1)) )
:          (derv (firstterm poly) var) ) )))
DERVEXP
:(dervprod '(mult x 2) 'x)
  (ADD (MULT X 0) (MULT 2 1))
:(dervexp '(exp x 2) 'x)
  (MULT (MULT 2 (EXP X 1)) 1)

```

We now have the entire system, so let's try some difficult stuff.

```

:(derv '(add (mult 3 (exp x 2)) 15) 'x)
  (ADD (ADD (MULT 3 (MULT (MULT 2 (EXP X 1)) 1))
        (MULT (EXP X 2) 0)) 0)
:(derv '(add (add (mult a (exp x 2)) (mult b x)) c) 'x)
  (ADD (ADD (ADD (MULT A (MULT (MULT 2 (EXP X 1))
        (MULT 1)) (MULT (EXP X 2) 0)) (ADD (MULT B 1)
        (MULT X 0))) 0)
:(setq a (derv '(mult (add x 1) (sub 1 x)) 'x) 'x)
  (ADD (MULT (ADD X 1) (SUB 0 1)) (MULT (SUB 1 X)
        (ADD 1 0)))
:(setq x 3)
3
:(eval a)
-6

```

Order and simplification are the first step toward the mastery  
of a subject -- the actual enemy is the unknown.  
--Thomas Mann

### Chapter 23: SIMPLIFYING POLYNOMIALS

Let us now return once again to the world of polynomials. We have only one task left before us, namely, the simplification of a polynomial. Remember from the last episode that the result of the DERV function could be rather messy, as the following aptly demonstrates.

```
:(deriv '(mult (add x 2) (add x 3)) 'x)
(ADD (MULT (ADD X 2) (ADD 1 0)) (MULT (ADD X 3)
(ADD 1 0)))
```

There is no intelligence in the DERV function. It should be clear that (MULT (ADD X 2) (ADD 1 0)) can be simplified to (ADD X 2). In fact, there are a whole bunch of similar simplifications that can be performed on polynomials. Here is our list:

LISP form	Simplified form
MULT ? 0	0
MULT 0 ?	0
MULT 1 ?	?
MULT ? 1	?
ADD 0 ?	?
ADD ? 0	?
SUB ? 0	?
EXP ? 0	1
EXP ? 1	?
EXP 0 ?	0
EXP 1 ?	1

[where ? is any LISP expression]

Assume we have at our disposal a function called SIMPLIFY which will perform these transformations. Here is some sample behavior:

```
:(simplify '(add x 0))
x
:(simplify '(exp 0 0))
1
:(simplify '(mult (mult 0 x) y))
0
:(setq p (deriv '(mult (add x 2) (add x 3)) 'x))
(ADD (MULT (ADD X 2) (ADD 1 0)) (MULT (ADD X 3)
(ADD 1 0)))
:(simplify p)
(ADD (ADD X 2) (ADD X 3))
```

When we apply the SIMPLIFY function to an expression of the form MULT 1 ?, the result should be the result of applying SIMPLIFY to ?. This means that SIMPLIFY is recursive [what else?].

The following function implements the above table:

```
(define (simplify1 (lambda (poly)
: (cond
:   ((null poly) nil)
:   ((atom poly) poly)
:   ((equal 'mult (function poly))
:     (cond
:       ((equal 0 (firstterm poly)) 0)
:       ((equal 0 (secondterm poly)) 0)
:       ((equal 1 (firstterm poly))
:         (simplify1 (secondterm poly)))
:       ((equal 1 (secondterm poly))
:         (simplify1 (firstterm poly)))
:       (t (list 'mult
:         (simplify1 (firstterm poly))
:         (simplify1 (secondterm poly))))))
:   ((equal 'add (function poly))
:     (cond
:       ((equal 0 (firstterm poly))
:         (simplify1 (secondterm poly)))
:       ((equal 0 (secondterm poly))
:         (simplify1 (firstterm poly)))
:       (t (list 'add
:         (simplify1 (firstterm poly))
:         (simplify1 (secondterm poly))))))
:   ((equal 'sub (function poly))
:     (cond
:       ((equal 0 (secondterm poly))
:         (simplify1 (firstterm poly)))
:       (t (list 'sub
:         (simplify1 (firstterm poly))
:         (simplify1 (secondterm poly))))))
:   ((equal 'exp (function poly))
:     (cond
:       ((equal 0 (secondterm poly)) 1)
:       ((equal 1 (secondterm poly))
:         (simplify1 (firstterm poly)))
:       ((equal 0 (firstterm poly)) 0)
:       ((equal 1 (firstterm poly)) 1))
:     (t poly) ))))
SIMPLIFY1
```

SIMPLIFY1 uses some of the help functions from the last chapter. The structure of the function follows rather closely the list of simplifications we gave above. Notice the recursive calls when the terms are not constant.



Let's compare it with our idealized SIMPLIFY.

```
(simplify1 '(mult (add x 2) (add 1 0)))
(MULT (ADD X 2) 1)
(simplify '(mult (add x 2) (add 1 0)))
(ADD X 2)
```

We have here a discrepancy. What is the source of the problem? Well, when SIMPLIFY1 simplifies (ADD 1 0), it gets 1 as it should. However, the test for multiplication by 1 has already been performed before this. SIMPLIFY1 can't make the second simplification. If we view the polynomial as a tree, then SIMPLIFY1 is moving down the tree and any reduction performed on subtrees can't migrate back to the upper levels. How can we beat this conundrum? What we really want to do is to keep applying SIMPLIFY1 to the polynomial until the application no longer results in any change. Let's write a function which goes around in a loop while continually applying SIMPLIFY1 until a final constant expression is reached.

In order to do this in LISP, we will need a new construct, the PROG. PROG is sort of hard to explain, so we will start with a simple example.

```
(define (counter (lambda (n)
:   (prog (m)
:       (setq m 0)
:       test (cond ((greater m n) (return nil)))
:       (print m)
:       (setq m (add m 1))
:       (go test)) )))
COUNTER
```

A PROG has several parts: a list of "local" variables, a sequence of LISP expressions, and some "branches" and "labels". In our example above, the list "(m)" immediately after the PROG is the list of local variables. These variables are just like LAMBDA variables in that they are in existence only inside of the PROG statement. Following the list of PROG variables is a sequence of LISP expressions. These expressions are either normal, everyday, garden variety expressions, a branch, a label or a RETURN. A label is an atom in the sequence of lists. Above, "test" is a label. A branch is a LISP expression of the form "(go @)", where @ denotes some label. When LISP EVALs a branch expression, control is transferred to the expression following the specified label. In our example, the expression "(go test)" transfers control to the label TEST. All other expressions transfer control to the following expression. A RETURN statement exits the PROG loop, and returns the value of its argument, here NIL. If you have a PROG loop, then you should definitely have a RETURN statement [probably in a COND expression] to get out of the loop, or you will loop forever!

Here is a sample run of COUNTER:

```
(counter 5)
0
1
2
3
4
5
NIL
```

```
:(counter 0)
0
NIL
```

Here is the PROG for simplify.

```
:(define (simplify (lambda (poly)
:  (prog (poly1)
:    loop (setq poly1 (simplify1 poly))
:      (cond ((equal poly poly1) (return poly)))
:      (setq poly poly1)
:      (go loop) )))
SIMPLIFY
:(simplify '(mult (add x 2) (add 1 0)))
(ADD X 2)
```

This function continues to simplify the polynomial until there is no change between two successive simplifications. It then returns the simplified polynomial.

Efficiency of a practically flawless kind may be reached  
naturally in the struggle for bread.  
--Joseph Conrad

## Chapter 24: EFFICIENCY AND ELIMINATION OF RECURSION

We have disregarded one issue throughout this entire book, namely, how hard the computer has to work to evaluate a function. In this chapter, we are going to look at some different ways of writing the same function, with emphasis on the efficiency of the evaluation.

Here are four different versions of the factorial function.

```

(define (fact1 (lambda (n)
:   (cond
:     ((equal n 0) 1)
:     (t (mult n (fact1 (sub n 1)))) )))
  FACT1
(define (fact2 (lambda (n)
:   (cond
:     ((equal n 0) 1)
:     ((equal n 1) 1)
:     (t (mult n (fact2 (sub n 1)))) )))
  FACT2
(define (fact3 (lambda (n)
:   (prog (m prod)
:     (setq m 0)
:     (setq prod 1)
:     loop (cond ((equal m n) (return prod)))
:     (setq m (add m 1))
:     (setq prod (mult m prod))
:     (go loop) )))
  FACT3
(define (fact4 (lambda (n)
:   (fact4A n 1) )))
  FACT4
(define (fact4A (lambda (n m)
:   (cond
:     ((equal n 0) m)
:     (t (fact4A (sub n 1) (mult n m)))) )))
  FACT4A

```

We haven't shown them working, but take our word for it, they do. What are the salient differences between each of the functions?

FACT1 and FACT2 are the standard recursive definitions of the factorial. However, since FACT2 tests for an argument of 1, it will end a chain of recursive calls one step sooner than FACT1. We still need to test for 0 because 0 is a special case. The importance of one less recursive call is, in this application, negligible.

FACT3 shows the factorial function in its iterative form. There is only one function call, but the function will loop  $n$  times just as FACT1 will call itself  $n$  times. Depending upon the

phase of the moon, the iterative solution might be more efficient for the computer [i.e. it will execute faster]. The recursive form will usually be more legible, though.

The fourth definition uses what is known as a **COLLECTION** variable, or an accumulation variable. As we decrement *n* we keep the running product in the collection variable *m*. The **FACT4** function serves only to pass the value of *n* and set up the collection variable for the function **FACT4A**. Although it may not seem very useful, this technique can be used to define very efficient recursive functions. It is particularly useful in cases where some values of the function are recomputed by different recursive calls. The collection variable can hold the temporary value and save some recursive calls.

So much for factorial. Several times before we mentioned that LISP was an interpreter. What does this mean? Language processors come in two different flavors, interpreters and compilers. An interpreter is a computer program written in assembly language (the language that is very close to what the computer understands directly). An interpreter works in what is called a **READ-EVAL-PRINT** loop. If we were to call a function 1000 times, LISP would re-evaluate each part of the function 1000 times. This is a waste of time. That is where a compiler comes in. A LISP compiler would translate each function into machine language (this is what the computer processor understands directly). This would make each function execute much more rapidly. You typically would not want to use a compiler to compile functions while you are developing the programs. Because compiling takes a finite amount of time, you normally would want to wait until all your functions are debugged before running them through a compiler.

## The P-LISP Tutorial -- ERRATA sheet and disk loading info

) Please note the following corrections:

Page 2-2 First line in last paragraph should have 'a' inserted after use.

Page 3-1 Reference to dotted pair should be omitted, since they have been removed from P-LISP ver 3.0

Page 13-2 Omit the sentence The value of STUFF is indicated by the "\*" in the above tree.

Page 13-2 Last paragraph ws should be is.

### Chapter 1

Version 3.0 of P-LISP does not allow ADD to take more than 2 parameters i.e. (ADD 3 4 5) was O.K. under the old version of P-LISP, but will give a too many args error in the new version. Several examples took advantage of this feature and must be corrected as appropriate.

### Chapter 16

There have been parentheses left out of the MAXLIST and INDEX functions. Here are the corrected ones:

```
:(define (maxlist (lambda (l)
:  (cond
:    ((null l) nil)
:    ((null (cdr l)) (car l))
:    ((greater (car l) (car (cdr l)))
:      (maxlist (cons (car l) (cdr (cdr l)))))
:    (t (maxlist (cdr l)))))
:  MAXLIST
```

```
:(define (index (lambda (a l)
:  (cond
:    ((null l) nil)
:    ((null (member a l)) 0)
:    ((equal a (car l)) 1)
:    (t (add (index a (cdr l)) 1)))))
:  INDEX
```

We apologize for the above errors. All the programs are correct on disk. If you find any additional errors, or just wish to give us any comments, please drop us a note.

To use the disk, boot up lisp normally BRUN LISP.

Use the LISP LOAD command to load the BOOK workspace :(LOAD BOOK)

The BOOK workspace may be on the BACK of the LISP disk.

Note that you must do this every time you try the examples from the book, since certain functions used in the book do not appear in standard P-LISP.



# The P-LISP Tutorial INDEX

ABS	9-2	FACT3	24-1	PARTS	21-3
ADD	1-2	FACT4	24-1	PROG	23-3
APPLY	20-1	FACT4A	24-1	Parenthesis	1-3
ASSOC	16-2	FACTORIAL	14-3	Pegasys Systems	P-1
Algorithm	18-1	FEXPRs	19-1	Predicates	5-1
Artificial Intelligence	P-1	FIRST	6-1	Predicates	1-4
Atoms	4-1	First Node	12-1	Property	21-1
BREAK mode	7-2	Formal Arguments	6-1 8-1	QUOTE character	2-3
BUILT-INS	7-1	Function expression	8-1	RAC	14-3
Behavior of formal args	6-2	GREATER	1-4	READ-EVAL-PRINT	24-2
Binary tree	12-1	Global Environment	15-1 15-2	RECITE	10-1
Branches	12-1	Grossis	P-1	REPLACE	16-2
CAAR etc.	2-4	HELP function	7-1 18-5	RETURN	23-3
CAR	2-1 2-3	INDEX	16-1	REVERSE	6-3
COR	2-1 2-3	INTERLISP	P-1	Recursion	10-1
COLLECTION variable	24-2	Interactive	1-3	Root	12-1
CONC	3-1 3-2	Joining Lists	3-1	Rules of FEXPRs	19-2
COND	9-1	LAMBDA	8-1	SIMPLIFY	23-1
CONS	3-1	LENGTH	17-1	SIMPLIFY1	23-2
Characters	4-1	LISP ,	P-1	SNOC	16-2
Conditional	9-1	LIST	7-3	SUB	5-2
DEFINE	8-1	LISTS	2-1 12-1	SUDR	21-5
DERV	22-2	Lambda Expressions	21-1	Scope Considerations	15-1
DERIVATION	22-3	Lambda-binding	8-3 15-1	Self-defining atoms	4-2
DERVEXP	22-4	Lambda-list	8-3	Side Effects	15-3
DERIVINUS	22-3	Leaves	12-1	Starting	1-2
DERVPROD	22-3	List Processor	P-1	Style	P-1 14-1
DERVSUM	22-3	Local Environment	15-2	Subexpressions	1-4
Defined atoms	4-2	Local variables	23-3	Suspended Evaluation	14-1
Defining Functions	6-1	MACLISP	P-1	Suspended functions	7-2
Differentiation	22-1	MAKELIST	7-2	TRACE	11-1
EQUAL	5-2	MAPCAR	17-1	Tag	21-2
EVAL	20-1	MAXLIST	16-1	Termination conditions	14-2 14-3
EVEN	17-1	MIT	P-1	Top Down Programming	18-4
EXP	16-1	MULT	1-3	Tracing functions	11-1
Efficiency	24-1	Maps	17-1	Tree Searching	13-2
Elimination of Recursion	24-1	McCarthy, John	P-1	Trees	12-1 13-1
Environment	15-1	Mid-evaluation	7-2	UNTRACE	11-1
Evaluation	2-2	NIL	2-1 2-3	Unevaluating Functions	19-1
Extending Lists	3-1	NUMBER	1-4	Uses of lists	3-3
FACT1	24-1	Overview	P-2	Values	4-1
FACT2	24-1	P-LISP	P-1	ZERO	1-5

