LEARNING LISP

This material has *NOT* been updated from the original 1984 text which I found it completely by accident in a tarball at this address: http://venus.deis.unical.it/manuals/llisp/index.html. There are *many* errors and typos, and the version of lisp (P-Lisp, which ran on the Apple II) predates commonlisp, and no longer exists. So I *strongly* encourage you *not* to actually use this to learn Lisp! -- Jeff Shrager, 20060430

The original tarball: [112 Kb]

Here's a new tutorial introduction to **LISP** (LIST Processor), the much-talked-about language of artificial intelligence.

Whatever your computer background, this comprehensive, clear-cut primer will teach you one of the oldest languages still in use--one that's simple and fun to learn. You'll become familiar with:

- **LISP's** basic data structure and functions
- how to define and edit your own functions
- trees and recursion
- advanced **LISP** programs
- and much more.

You'll also find concrete, elementary examples that will help you grasp more abstract ideas, plus exercises to reinforce what you learn in each chapter. Included as a special feature is a sample dialogue with ELIZA, Joseph Weizenbaum's classic **LISP** program.

Stop wondering what LISP is all about and what it can do for you. Let **Learning LISP** teach you everything you need to know about a language that will make your computer a valuable partner in thinking.

Gnosis, a Philadelphia-based software company, currently markets a **LISP** interpreter. The interpreter, **P-LISP**, is in use by hundreds of colleges and research facilities around the world for research, artificial intelligence development, and teaching.

Contents

Preface

1. Getting Started
2. Lists, CAR and CDR
3. More Lists
4. Atoms and Values
5. Bag of Predicates
6. Defining Your Own Functions
7. Help Functions
8. How to Save the World
9. This Thing Called Lambda
10. The Conditional
11. Simple Recursion
12. The Lisp Editor ED
13. Lists as Trees
14. Trees and Recursion
15. A Style of Programming
16. Scope Considerations
17. Maps
18. Isplay Ogrammingpray
19. FEXPRS: Unevaluating Functions
20. Control Structures
21. Eval and Apply
22. Properties and Lambda Expressions
23. Differentiating Polynomials
24. Simplifying Polynomials
25. Efficiency and Elimination of Recursion
26. ELIZA
27. The P-Lisp Interpreter

Appendix: The Lisp Editor

index

Jeff Shrager (jshrager@stanford.edu)

Co-Founder and Chief Technology Officer, CollabRx
Associate Professor, Stanford University, Symbolic Systems Program (consulting)

"Correlation does not prove causality,
but they are highly correlated."

(Photo: Marty Hellman; The glider is a Stemme S10-VT)

Personal Info:

- My vita (This is a direct link to my publications list)
- Talks (etc) available online as powerpoint
- Supplementary materials for my papers

"The earth does not, in fact, move around the sun
any more than the sun moves around the earth.
The former point of view just simplifies the math."

I am co-founder and Chief Technology Officer of CollabRx. CollabRx is a startup that builds and operates virtual biotechs.

At Stanford I teach several courses, including In Interaction Analysis (Symbolic Systems 145) which focuses on human learning about and interaction with complex engineered systems, and **Symbolic Biocomputing** (Symbolic Systems 216) where we study Artifical Intelligence applications in biological computation. (Symbolic Biocomputing is no longer offered as a formal course. Instead, the materials are online and open source in the form of BioBike Live Tutorials.)

Various Projects:

It is not merely from the ground evidence itself,
but primarily through the progressive accumulation
of justified interpretations of that evidence
that sound conclusions are reached.

I am PI on the *BioBike* project. (Formerly called "BioLingua") *BioBike* is a web-based programmable biological knowledge-base, which went to live public alpha testing in August 2003. It is built on top of BioLisp, which is, in turn, built on top of Common Lisp. *BioBike* is a complete knowledge-based computational biology resource, enabling biologists to manipulate biological knowledge and data, and providing a platform for computer scientists working on methods in computational biology to develop their methods and deploy them immediately to working biologists. If you'd like to experiment with the *BioBike* Multi-Cyano programmable knowledge base, drop me an email and I'll be happy to give you an account. The documentation root for this *BioBike* instance is here. (I am the co-founder, editor, and webmaster of BioLisp.org, a site dedicated to intelligent applications in BioComputing.)

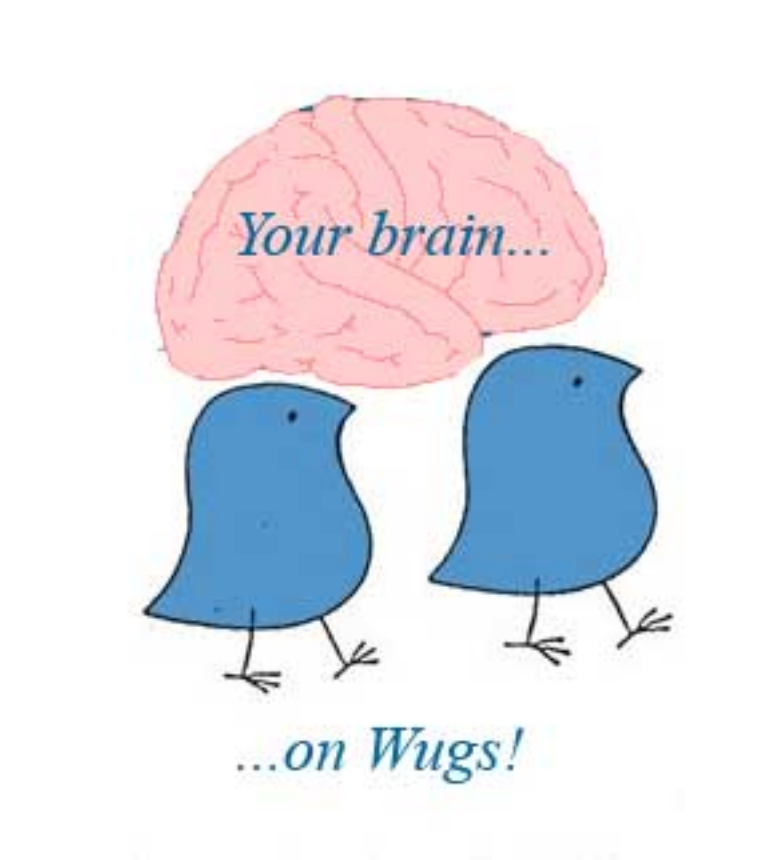
Mnemotheque is a personal exploration in interactive multi-media memorial, developed by my sister, Monique, and me in order to memorialize our family's history with The Holocaust.

Gorilla Science (Education in the Wild) is an educational activity in which students produce public installations (usually posters) that explain the role played by scientific principles in every-day domains. This links is a large (~3M) pdf file containing various of the early Gorilla Science documents, including some of the installations designed by Mary Burns' eight graders around 1995. (Conception: Jeff Shrager and Kevin Crowley, University of Pittsburgh Learning Research and Development Center; Early Art: Christen Napier; K-12 Implementation: Mary Burns and the students of Franklin Regional Junior High School in Westmoreland County, PA.) Gorilla Science is making inroads in Pittsburgh; Check out the *Explanatiods* project!

Fun Stuff:

Check out these videos of my sister Monique's band, *fmz*, in their debut concert at the Havana Cafe in Toulouse, France:

- fmz: take my mind (live)
- fmz: spin off (live)
- fmz: Death On TV (studio rock video)



The time has come the walrus said to talk of many things,
Of diastolic pressure and of diuretic flings,
And whether venous pressure falls with digitalization,
When the failure's on the distaff side of the circulation,
And if ethacrynic acid is the answer to our prayers,
How come we still use morphine when the diuretic's there?
Indeed, the greatest mystery of pulmonary edema,
Is why the patients do so well without a decent schema!

-- Anon; ~1971
From a lecture on accute pulmonary edema

(with apologies to Lewis Carroll)

A fun glider aerobatics video, made with Randy Gobbel, Melissa, and Rex Mayes on Feb. 14, 1989. [~15MB]

(I recently realized that I have been involved in teaching bioinformatics for a *very long* time! Here is a paper on computational protein secondary structure prediction by some of my students from the Pennsylvania Governor's School for the Sciences, a program for bright *high school students* that took place at CMU in *1983*! The program is even in Lisp! (The paper says that you can contact me for the code; Good luck; It's probably on *mag tape* someplace!))

PreCognitive Science Online (A parody of the new EJournal *Cognitive Science Online*)

The Arnon-Calvin Challenge: A Turing Test for Computational Systems Biology

Diary of an Insane Cell Mechanic: Between May and December of 2000, when I started fulltime at The Carnegie, I kept a cognitive diary about what it's like to become a molecular biologist.

Filer (for Unix) is a regular expression-like meta-command creator -- a sort of combination of ls, sed and xargs, but with a significantly simpler pattern language. Filer is protected by the GNU Public License.

Stories: A small collection of badly written, but possibly amusing, short stories about diving, flight, the aurora, and other random experiences.

- Natural Thing (2001) -- A safari to North Pole, Alaska to see the Aurora.
- Sleepless in the Bat House (??) -- In which I join Betsy on a trek across eastern Australia in search of bat spit.
- Reena (1989) -- A glider off-field landing and the little girl who loves birds.
- Abe (1994) -- Diving, hospitals, decompression chamber rides, and volunteer firefolk.
- Bat Girl (~1995) -- A night out in the Pennsylvania hills with a bat biologist.
- Blind Flight (1994) -- Instrument flight -- a student's-eye view.
- First Solo (~1986) -- All pilots are required to write this story! (It's in the FARs!)
- Tracy (1991) -- A fictional daughter-to-father letter with a twist.

Quotable: (Quotes embedded in the page, above are my own.)

> I said, 'How dyou do that kynd of gethering what youre going to do? Do you all set down and pul datter or dyou jus think to gether or what?'

He said, 'We do some poasyum.'
I said, 'Whats poasyum'
He said, 'It aint jus poasyum you all ways say some poasyum. You ever seen a nes of snakes?'
I said, 'Yes.'
He said, 'I never but 1 of the hevvys tol me they do the same theywl get all in a tangl slyding and sqwirming and ryving to gether. Which is how we do it all the many rubbing up to 1 a nother skin to skin and talking vantsit theary. Which is a kynd of hy telling and trantsing. Thats when the singing and the shouting come the many cools of Addom and the party cools of stoan. The strong and the weak inner acting and what happent in the cloudit chaymber.'
I said, 'Is that where the seed of the red and the seed of the black come in to it?'
He said, 'Yes, howd you know that?'
I said, 'When you ben having your fit you ben talking vansit theary. If you cud do it then and you can do it now may be you dont even nead to gether may be you can get them Nos. oansome.'

-- Russell Hoban, *Riddley Walker*

"False facts are highly injurious to the progress of science, for they often endure long; but false views, if supported by some evidence, do little harm, for everyone takes a salutary pleasure in proving their falseness: and when this is done, one path toward error is closed and the road to truth is often at the same time opened."

-- Charles Darwin, 1871, *The Descent of Man and Selection in Relation to Sex*

"I had a feeling once about Mathematics, that I saw it all---Depth beyond depth was revealed to me---the Byss and the Abyss. I saw, as one might see the transit of Venus---or even the Lord Mayor's Show, a quantity passing through infinity and changing its sign from plus to minus. I saw exactly how it happened and why the tergiversation was inevitable: and how the one step involved all the others. It was like politics. But it was after dinner and I let it go!"

-- Winston Churchill, 1930, *My Early Life: A Roving Commission*

"*Dopeler effect* (n): The tendency of stupid ideas to seem smarter when they come at you rapidly."

-- From a contest in *The Washington Post*(2001) asking readers to make up new words that are similar to existing words ... but funnier.

"The name of the game now is 'modelling.' A lot of it I can't see for sour owl shit. How

can your write or talk authoritatively about something if you haven't seen it?"

-- Field Geologist David Love, quoted in John McPhee's *Annals of the Former World* (1998)

"Creationist Method. Creationists believe that man was instantaneously created by God based on an account in a book called 'the Bible.' Several thousand years ago, a small tribe of ignorant near-savages wrote various collections of myths, wild tales, lies, and gibberish. Over the centuries, these stories were embroidered, garbled, mutilated, and torn into small pieces that were then repeatedly translated into several languages successively. The resultant text, creationists feel, is the best guide to this complex and technical subject [how humans evolved]."

-- Tom Weller, 1985, *Science Made Stupid*

---

cllib

LEARNING LISP

Contents | Getting Started

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 some of the later examples easier to understand but is by no means required for the rest of the book.

The book was written originally by Jeff Shrager (currently at Carnegie-Mellon University) and Steve Bagley (currently at MIT) while they were undergraduates at the University of Pennsylvania. The Moore School Computing Facility, in the School of Engineering and Applied Science, at the University of Pennsylvania supplied computer time so that it could be developed online. Additional material was contributed by Stewart Schiffman of the Gnosis staff, and by Steve Cherry, author of P-LISP, which is the dialect of Lisp that the examples in this book use.

WHY SHOULD YOU LEARN LISP?

Lisp is important. Lisp is one of the oldest languages still in active use. It was invented for and is still used (workshipped) by computer scientists 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 mechanism 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 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 1950s 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 1960s several other versions were developed across the country for various different machines. MacLisp from MIT, InterLisp, formerly BBN Lisp, and MTSLisp from the University of Michigan are three currently available.

The dialect spoken in this book is P-Lisp. This is yet another Lisp system, but this one runs on the Apple and various other microcomputer systems. Personal Lisp computers are a net innovation in the computer world. MIT has big microcomputers that run a MacLisp derivative. Although P-Lisp isn't as powerful as those microcomputers (primarily because the computers it runs on are much smaller), the ideal of having your own Lisp processor remains. In particular, this version of the book goes with P-Lisp version 3.1. If you don't have that version, some minor details will be different (for example, there may not be any floating point arithmetic).

If you are using another dialect of Lisp, you shouldn't have too many problems. In most of the examples, we use a fairly common subset of Lisp functions.

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 the difficulties associated with some of the more abstract concepts in Lisp by working up to them from elementary, concrete examples. We suggest that you carefully follow through all the examples presented. Access to your computer is desiderable 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. This book was not meant to be read in one sitting, so take your time. There are a few problems at the ends of some of the chapters. Do them if you feel like it. Some of them aren't meant for solution as much as for thought, so if you think about them rather than actually doing them, that's sufficient.

TYPOGRAPHIC CONVENTIONS

Sometimes (especially in longer examples) in this book you will find that 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. If this convention is in use, there is one minor exception to its rule. That is, the letter "L" will always appear in upper case because its lower case form is the same as a one and this might be very confusing in a program example. Unfortunately, L is very often used in Lisp programs to mean "List".

Parenthetical remarks are going to be enclosed in square brackets ["[]"] instead of the normal

parentheses because Lisp makes a lot of use of parentheses and things will get confused.

THE BOOK DISK

This texti is available with a floppy disk that contains all the functions that we use in this book. Its purpose is to save you typing time and to give you a pre-created environment to write your programs in. If you already know what a Lisp environment is, fine. If not, don't worry about it. It's covered later in the book.

OVERVIEW

The format of the book is as follow: 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. We lay out a few examples of complicated Lisp programs in detail. The last section of the book consists of some chapters on advanced Lisp techniques, and the guts of the Lisp system itself.

Enjoy, and please feel free to let us know about any problems you have or other things you would like to see.

Contents | Getting Started

LEARNING LISP

Contents | Preface | Lists, CAR and CDR

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 Lisp up and running [see the explanation of how to do this in the *P-Lisp Manual*]. You should see the following at the top of the screen:

```
        GNOSIS INC.
     P-LISP   VER. 3.1.2
     -------------------

COPYRIGHT 1982 BY STEVEN CHERRY
     ALL RIGHTS RESERVED
```

When you see this display it means that you have successfully entered Lisp. The ":" prompt that you see on the last line typed by the computer means that Lisp is waiting for you to type something in. You may type in what you wish. After you hit the RETURN key 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 or READ-EVALUATE-PRINT much later on.

Note that if you hit RETURN several times, each time 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 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 Lisp we use the ADD function. We add 1 and 2 by typing:

```
:(ADD 1 2)

  3
```

Yeah! Lisp can add. What actually happened? 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].
- Lisp responded immediately with the answer. Lisp is an interactive system, and it will always display the answer immediately, unless you tell it otherwise. Later we will see how to tell it otherwise.

Let's try some more addition.

```
:(ADD 12 3

:)

  15

:(ADD 1)

    ** ERROR: TOO FEW ARGS **
      ADD :: (1 )

+()

  NIL

:(ADD 11 8 3)

    ** ERROR: TOO MANY ARGS **
      ADD :: (11 8 3 )
```

```
+()

  NIL
```

Here we first try the same example but we have forgotten the closing parenthesis. Lisp is waiting for that closing parenthesis so it comes back with a ":" prompt. We enter the closing parentheses, and now 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 line shows something a little funny. We asked Lisp to add up one number. Because adding up just one number is not particularly meaningful or useful, Lisp returns an error message that there are too few numbers to add. 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, simply type "()". We will deal with this mode of operation later.

The last line shows what happens if you try to add up three numbers--the same sort of error! Well, there is no penalty for mistakes [we won't tell]. It makes a little more sense to add up more than two numbers than it did to add up just one number. We'll see much later that we can actually fix ADD ourselves to do this [or any other sort of behavior that we like].

The first thing in a pair of parentheses is the *function* name and the things after that are the *arguments* [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.

Besides addition, Lisp can also perform multiplication. The name of the multiplication function is MULT. Let's try it out!

```
:(MULT 2 3)

  6

:(MULT 9 2)

  18

:(MULT 1 2 3 4)

    ** ERROR: TOO MANY ARGS **
      MULT :: (1 2 3 4 )
```

```
+()

  NIL

:(MULT 1.2 4)

  4.8

:(MULT 2 (ADD 1 2))

  6
```

The first two examples reassure us that Lisp can, in fact, multiply.

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. Again, we type "()"" to get back to the normal colon prompt.

The next example shows that Lisp will deal with non-integers. Floating point math is nice, but not critical because, as we will soon see, Lisp's strength does not lie in arithmetic.

The last of the above lines is the most interesting. Lisp tries to perform the MULT function but finds that in place of the second argument is a *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. 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 the 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 2 1) (ADD 1 3)))

  24

:(MULT 1 (MULT 2 (MULT 3 (MULT 4 1))))))))))

  24
```

One important thing to notice about these examples is that in the last one there were too many closing

parentheses. This is fine and, in fact is very handy sometimes when you lose count. All you need to do is keep typing lots of closing parentheses and eventually you'll get back to the colon prompt.

Now for one more 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 number, "NIL", otherwise. Obviously the word "seven" is characters [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.

Finger Exercises

Practice starting Lisp and typing in expressions. You might actually take the time to do all of the examples in this chapter. Also, do some math and make sure that Lisp can do math as well as you can. See if you can come up with a way of changing algebraic expressions into the equivalent Lisp mathematical expressions.

Contents | Preface | Lists, CAR and CDR

LEARNING LISP

Contents | Getting Started | More Lists

Lists, CAR and CDR

We are 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 obkects 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.

You should note that the parentheses are used to denote the list; they are not actually part of the list.

"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 8 3)

  11
```

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 :: NIL
```

```
+()

  NIL
```

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

```
:(ADD (1 2))

    ** ERROR: BAD ATOMIC ARG **
      EVAL :: NIL

+()

  NIL
```

Compare the expressions (ADD 1 2) and (ADD (1 2)). In the first one, the ADD function acts on two separate 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). Remember that Lisp first evaluates the arguments before applying the function.

When Lisp encounters (ADD (1 2)), it first tries to evaluate the argument to ADD, namely the list (1 2). Note that "1" is not a Lisp function. [Remember, if Lisp is trying to evaluate a lista, 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 lists 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. We're going to begin applying the quote quite liberally from now on. Be very careful to watch what does and does not get evaluated.

```
:'(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. Manipulating 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 complement 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 a 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))

  ADD

:(CDR '(ADD 1 2))

  ( 1 2 )
```

As we mentioned a little earlier in this chapter, the expressions that we are typing into Lisp are lists, just as "(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" [NOTE: Lisp aficionados claim to be able to pronounce all 28 combinations of CAR and CDR]. For example, the expression (CADDR ANYLIST) is the same as the longer expression (CAR (CDR (CDR ANYLIST))). This book will not use these too much, but you should be familiar with them since many things written in Lisp do use them. The above example

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

could have been written

```
:(CDADR ' (() ((BOZO) (NO NO))))

  ( (NO NO ) )
```

Exercises: Car For Yourself

We still aren't deep enough into Lisp to do any entertaining or interesting exercises so your task is to make up some exercises for this chapter and do them.

Contents | Getting Started | More Lists

LEARNING LISP

Contents | Lists, CAR and CDR | Atoms and Values

More Lists

In the previous chapter we learned all about taking lists apart. 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. These names are a little more reasonable than CAR and CDR, but not much.]

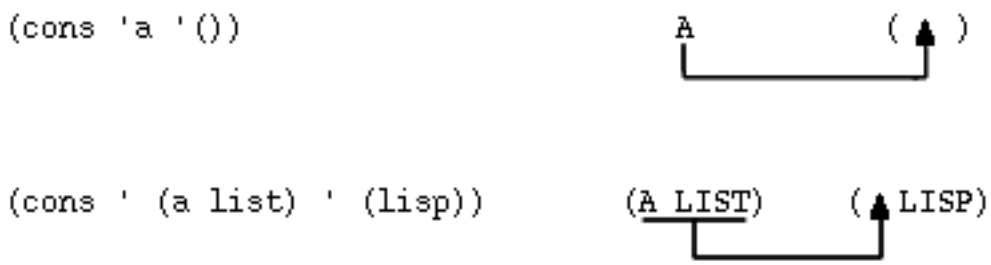
Let's first play with the CONS function.

```
:(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. Don't worry about dotted pairs for now; Lisp will print them, but they are not used very often, nor are they very important.

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 is very important. You should make sure that you throughly 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 first 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

:(CON '(B C) 'A)

    ** ERROR: BAD LIST ARG **
      CONC :: ((QUOTE (B C ) ) (QUOTE A )
      )

+()

  NIL
```

Moral of the story: CONS can deal with atoms, CONC can't. CONS is the opposite of a CAR and a CDR.

```
:(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, and be 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 since 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

```
(ADD (MULT 12 (ADD x 2) ) 17)
```

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.

Exercises

Aha! Now we know enough to do something of interest:

1. Given the phonebook data structure mentioned above, write an expression which will make a list with those name/number pairs in reverse order. That is: ((jane 2019999) (simon 5551212) (bill 1234567)). Remember to quote things correctly. You'll have to repeat the list a lot, so you may want to try a smaller example.
2. Do the same thing as the previous exercise. Make it come out the same way but with the names and numbers exchanged.

Answers

1. Let L represent the phone list

   ```
   ((BILL 1234567) (SIMON 5551212) (JANE 2019999))
   ```

   An expression to reverse this list is

   ```
   (CONS (CADDR L) (CONS (CADR L) (CONS (CAR L) '())))
   ```

2. We want to do the same thing we did for (1) except reverse each entry. The expression to reverse all three entries and the entire list is:

   ```
   (CONS
     (CONS (CADR (CADDR L)) (CONS (CAR (CADDRL)) '()))
     (CONS
           (CONS (CADR (CADR L)) (CONS (CAR (CADR L)) '()))
           (CONS
                 (CONS (CADR (CAR L)) (CONS (CAR (CAR L)) '()))
                 '())
           )
     )
   ```

Contents | Lists, CAR and CDR | Atoms and Values

LEARNING LISP

Contents | More Lists | Bag of Predicates

Atoms and Values

In the previous chapters we discussed lists of objects. These objects could have been either atoms or lists. The exact meaning of atom was sidestepped. 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, as follows.

- An atom can be any number the computer understands. This is called a numeric atom. These are *integers* like: 145, -15, 0, etc., or *floating point numbers* [ones with fractional parts] like: 1.4, -56.3, etc.
- 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 alphanumeric atom.
- The form of NIL "()" can be an atom.
- Alphanumeric atoms can have some funny characters in them [such as "*" and "+"] but special Lisp characters cannot be used in atom names. This should be clear by now. The characters "(", ")", 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:

```
:LOVE

    ** ERROR: UNDEFINED ATOM **
      EVAL :: LOVE

+()

  NIL

:(SETQ LOVE 5)

  5

:LOVE

  5

:(SETQ HAPPINESS LOVE)

  5

:HAPPINESS

  5

:(SETQ POWER FREEDOM)

    ** ERROR: UNDEFINED ATOM **
      EVAL :: FREEDOM

+()
```

```
  NIL
```

The first example just shows that, in fact, the value of love 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

:(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, 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 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 SANDWITCH '(SAND WITCH))

  (SAND WITCH)

:(CDR SANDWITCH)

  (WITCH )

:(CDR (CDR (CDR  SANDWITCH)))))

  NIL

:(CAR (CDR SANDWITCH))

  WITCH

:(CAR (CAR (CDR (SETQ CLOWNY ' (() ((BOZO

:) (NO NO)))))))))
  (BOZO )

:(CDR (CAR (CDR CLOWNY)))

  ((NO NO ) )

:(CAR (CAR (CDR (CAR (CDR CLOWNY))))))
```

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 thiss mumbo-jumbo. Remember that the arguments of a function are evaluated first and the results are 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)))))

  NO
```

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.

One last thing before we move on. It is often useful to be able to put funny characters [like spaces and open or close parens] into atom names. The most obvious use of this is to print things out. Even though it's technically against the rules to use spaces, etc., in atom names, Lisp provides a way to do so. The quotation character ["] can be used to enclose *atom* names. Don't confuse this with the quote that stops evaluation. For example:

```
:(SETQ "A LONG ATOM NAME" 5)

  5

:"A LONG ATOM NAME"

  5

:(SETQ "ANOTHER NAME" '"A MESSAGE")

  A MESSAGE

:"ANOTHER NAME"

  A MESSAGE

:
```

In the second example above, we quoted an enclosed atom in order to put a message with spaces in as the value of the atom: A LONG ATOM NAME
Note that Lisp doesn't print out the quotation marks around enclosed names. This is convenient because it permits us to use them as messages. You'll see this used if you look over the code for the Lisp editor that is included in an appendix. Don't look now, though. You'll need to know more of the language first.

Exercises

1. Indicate what you think Lisp would respond with if you type in the following:
    a. :(setq apple 'fruit)
    b. :(setq pear apple)
    c. :(setq iq 140)
    d. :(setq 15 12)
    e. :'apple
    f. :apple
    g. :"apple"
    h. :'"()"
2. Remember all the recopying that we did in the most recent set of exercises. Well, now you should be able to figure out how to set the phone list as the value of some variable, and just use that variable. Try it.

Answers

a. FRUIT
b. FRUIT
c. 140
d. You should expect this to cause an error; since numeric atoms are self-defining, you shouldn't be able to change their values.
e. APPLE
f. FRUIT
g. FRUIT. Note that APPLE and "APPLE" denote the same atom, since the atom names are the same (double quotes enclose atom names, but are not part of the name).
h. (). This creates an atom whose name is the characters "(" and ")". It is NOT the same as the atom NIL.

Contents | More Lists | Bag of Predicates

LEARNING LISP

Contents | Atoms and Values | Defining Your Own Functions

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, testing 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.

```
:(NULL (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 ARGS **
      EQUAL :: ((QUOTE SOLIPSIST ) )

+()

  NIL

:(EQUAL 'THREE 'FOR 'ALL)

    ** ERROR: TOO MANY ARGS **
      EQUAL :: ((QUOTE THREE ) (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.

Contents | Atoms and Values | Defining Your Own Functions

LEARNING LISP

Contents | Bag of Predicates | Help Functions

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 the system. 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 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)

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 argument, which in the FIRST definition is L, takes on the value of the actual argument 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

:L

    ** 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
```

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.

As with 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 out work. Applying meaningful names to simple things is one very important use of DEFINE.

Exercises: Rolling Your Own Functions

1. Remember the infamous phone list? Define FIRST, SECOND, and THIRD which will get the first, etc. entry from the phone list. Note that these are no longer valid phone lists themselves.
2. Now define two functions: NAME and NUMBER which give you the name or number from one of the result lists of the functions from the previous problem.
   You should be able to do:

(NAME (FIRST PHONELIST))
(NUMBER (FIRST PHONELIST)) etc.

3. Redo problems 1 and 2 from Chapter 3, using the functions you've just defined.

Answers

1.
```
(DEFINE (FIRST (LAMBDA (L)
        (CAR L))))

(DEFINE (SECOND (LAMBDA (L)
        (CADR L))))

(DEFINE (THIRD (LAMBDA (L)
        (CADDR L))))
```

2.
```
(DEFINE (NAME (LAMBDA (L)
        (CAR L))))

(DEFINE (NUMBER (LAMBDA (L)
        (CADR L))))
```

3. An expression to reverse the list is:

```
(CONS (THIRD L) (CONS (SECOND L) (CONS (FIRST L) '() )))
```

An expression to reverse the list and each entry is:

```
(CONS
  (CONS (NUMBER (THIRD L)) (CONS (NAME (THIRD L)) '()))
  (CONS
   (CONS (NUMBER (SECOND L)) (CONS (NAME (SECOND L)) '()))
   (CONS
    (CONS (NUMBER (FIRST L)) (CONS (NAME (FIRST L)) '()))
   '()))
    )
)
```

Contents | Bag of Predicates | Help Functions

LEARNING LISP

Contents | Defining Your Own Functions | How to Save the World

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 Lisp is that the latter are defined automatically each time you enter 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
```

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 **
      CONC :: ((FIRST L ) (LAST L ) )

+L

  (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. At this time, the formal argument "L" 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! The following is an example:

```
:(CONC 'GRATEFUL 'DEAD)

    ** ERROR: BAD LIST ARG **
      CONC :: ((QUOTE GRATEFUL ) (QUOTE
      DEAD ) )

+()

  NIL
```

It looks like we'll have to figure out some way of making a list with the results 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 put 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
```

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. It's better to have a lot of help-functions than not enough.]

```
:(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 Lisp already has a function called LIST which does exactly what MAKELIST does. We did it ourselves just for practise.

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 definition in the last chapter, they can help you to organize your thoughts by naming mental jobs into understandable parts.

Contents | Defining Your Own Functions | How to Save the World

LEARNING LISP

Contents | Help Functions | This Thing Called Lambda

How to Save the World

Now that you are beginning to put your own functions into the system it would be useful to learn something about how to save them for use later on. This will become especially important when there are big functions that you won't want to retype.

When you are using Lisp, everything that you do goes into a big pile called the environment. The environment holds all the functions that you've defined and all the atom values that you SETQed. As it turns out, the environment also contains any typos that you made, but we'll talk about these much later on. Normally, you should always save the whole environment together. That is, all the functions in a given environment are related, so you really want them all saved en masse.

The SAVE function takes everything in the environment and writes it out to a disk. You simply say,

```
:SAVE ENVNAME
```

and Lisp will dump everything you've done since you started to run Lisp, onto a file on the disk called ENVNAME. Notice that you don't need quotes here. SAVE is special in this way [like the first argument in SETQ]. Note also that you don't need parentheses. This is true only for P-LISP and not for most other Lisps that you may encounter.

The disk will do a few moments worth of clicking. Restart Lisp and type:

```
:LOAD ENVNAME
```

and sure enough everything is back exactly the way you left it! Both SAVE and LOAD have a lot of work to do, so you have to wait while they think about it.

By the way, although we used ENVNAME in the above example, you can call your environment anything you like. Any atom name can also be the name of the environment.

Microcomputers are not very reliable sometimes [nor are big computers]; so you might want to save your environment once in a while just for good measure. Then, if the computer dies for some reason, all you have lost is the work that you did between the last SAVE function and the time that it died. A LOAD command will restore you to exactly where you were at the last save.

Another technique that is handy in case of a problem is called *save alternation*. That is, don't keep saving into the same file. Suppose we were working along and decided that this was a good time to save the environment. We could start by saving into, for example, ENV-A. After a little while longer, we will want to save again, but this time we'll save into ENV-B. This leaves ENV-A intact in case there is a serious problem that could ruin ENV-B (for example, a cup of coffee is spilled on the disk), and we have ENV-A as a fallback. The next save, now our third, goes into ENV-A again, thus leaving ENV-B as the backup. Then back to ENV-B, and so on.

By alternating the saved names in this way you can almost guarantee that you won't lose everything, especially if the two files are on different disks. Be sure not to lose track, however. You wouldn't want to load ENV-A when the last file that you saved into was ENV-B or vice versa.

Another useful idea is to *time-date stamp* or *version stamp* the file names. That is, add some numbers to the environment name so that you can tell when it was saved. Here are some suggested forms:

ENVNAME-JAN-16-2PM
ENVNAME-1, ENVNAME-2, and so on
ENVNAME-01-16-82*2PM

Don't make the format too hard to remember, and be sure to delete very old versions so the disks don't fill up with garbage. By doing this, you have the advantage of not having to remember what the most recent version was; the filename remembers it for you!

Now that you know how to save your work, you might want to make a special "play" environment to keep everything that you have been playing with in. Also, you may want an environment for each major project that you write in Lisp. In a few more chapters we'll talk about how to manage the material inside the environment, but for now, managing the environments themselves is sufficient. Now back to more Lisp fun.

Contents | Help Functions | This Thing Called Lambda

LEARNING LISP

Contents | How to Save the World | The Conditional

This Thing Called Lambda

All the functions that were defined previously had something 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 cause 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 previous chapters, 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, the value of L would still have been available after the CAR operation. It seems that the formal arguments got SETQed to match the supplied argument when the function was invoked. The only difference is that after the function finishes [that is, the function-expression is done evaluating] the values get unSETQed.

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. This 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 to put two variables in the part of the expression supplied with arguments. 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 "LOVESICK" and "MOOSE", as long as we did the same in the function-expression also.

As an example of a two-argument function, here is a function that concatenates two lists together. We are calling it MERGE:

```
:(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 be bound [CONC is short for CONCatenate] to the first name "L"

and the second would be bound to the name "m".

```
( (a b c) (d e f) )
-------------------
(    L       m    )
```

Let's try giving Lisp nothing to connect to the formal argument:

```
:(FIRST)

    ** ERROR: TOO FEW ARGS **
      FIRST :: 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 the 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 an important concept in 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.

Contents | How to Save the World | The Conditional

LEARNING LISP

Contents | This Thing Called Lambda | Simple Recursion

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 as LIST1. If the result is not a NIL, then COND will evaluate the remaining things in the list, and will return the value of the last thing it evaluated. If the result is NILL, 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 CARs 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.

```
:(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, return 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 )

:
```

Exercises

1. Write a function that returns T if its first argument is less than or equal to its second argument, and NIL otherwise.
2. Write a function that compares two 2-atom lists, returning T if both lists are the same (assume

EQUAL only works on atoms).

Answers

1.
```
(DEFINE (LEQ (LAMBDA (N1 N2)
  (COND
   ((EQUAL N1 N2) T)
   ((GREATER N2 N1) T)
   (T NIL)
  )
)))
```

2.
```
(DEFINE (COMPARE (LAMBDA (L1 L2)
  (COND
    ((EQUAL (CAR L1) (CAR L2))
     (COND
       ((EQUAL ((CADR L1) (CADR L2)) T)
       (T NIL)
     )
    (T NIL)
  ))
)))
```

Contents | This Thing Called Lambda | Simple Recursion

LEARNING LISP

Contents | The Conditional | The Lisp Editor ED

Simple Recursion

Recursion occurs 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 and as such will know the answer without having to recur.

Let's try a simple program, called RECITE, to print out all the elements in a list with 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 because we are going to pass the CDR of the list, the list will get smaller with each recursive call.

Recursion is useless unless we can make it stop. RECITE, therefore, should do the following: If its argument is NIL, then it should return to NIL. [This type of test is called a *termination condition*. We need it to keep List 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.

```
:(DEFINE (RECITE (LAMBDA (STUFF)

:   (PRINT (CAR STUFF))

:   (RECITE (CDR STUFF))))))

  RECITE

:(RECITE '(THIS IS A TEST))
  THIS
  IS
  A
  TEST
  NIL
  NIL
  NIL
  NIL
  NIL
  NIL

+()

  NIL
```

{We hit control C.}

The list of NILs in the above execution will go on forever. We've cut off at seven in order to preserve our forests. This is a good time to learn about how to do that--that is, stop a function that is running wild.

The answer is control-C. When a Lisp function starts repeating, you simply hold CONTROL and hit C. This causes Lisp to break the function, suspend it, and enter the "+" mode. [We've talked about this before.]

Onward to another example. The function we are going to define is called MEMBER. This function will take two arguments, an atom and a list. MEMBER will return a T if the atom is one of the top-level elements of the list, NIL otherwise. We now exhibit the function definition and some examples of its uses.

```
:(DEFINE (MEMBER (LAMBDA (A L)

:         (COND

:           ((NULL L) NIL)

:             ((EQUAL 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. If it is, 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 of the list. Note that this process is guaranteed to terminate because 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.

Let's follow the MEMBER function with a debugging tool called TRACE. The Lisp TRACE function will tell us who calls whom and what is returned. When you see "-->>", it means that the function is being called. When you see "<<--", it indicates that the function is returning. Follow these examples

and watch what's happening. Note that you get an extra set of parentheses around the arguments in the "-->>" trace.

```
:(TRACE MEMBER)

  T

:(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
```

Note that the "T" result is passed back through each level of the 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 a Lisp function that returns the values to the routine that called it, never back to the user directly. We saw this in the ENDS example and it also applies to recursion.

The opposite of TRACE is UNTRACE.

```
:(UNTRACE MEMBER)

  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 tracing off of all of your functions at once, simply type (UNTRACE).

```
:(UNTRACE)

  NIL
```

As a third example, we will look at a recursive mathematical function, namely, the factorial. Recall that the factorial of n is the product of the first n integers and is defined by the following recursive formula:

```
n! = 1            if n = 0
    n*(n-1)!      if n > 0
```

Notice that the termination condition is already specified in the definition, namely, that the recursion stops when n=0. The factorial can easily be defined in Lisp, as follows:

```
:

:(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
```

It would be instructive to trace an evaluation of FACTORIAL to see how it works. Here's a trace of (FACTORIAL 5).

```
:(FACTORIAL 5)
     ->> FACTORIAL :: (5 )
     ->> FACTORIAL :: (4 )
     ->> FACTORIAL :: (3 )
     ->> FACTORIAL :: (2 )
     ->> FACTORIAL :: (1 )
     ->> FACTORIAL :: (0 )
     <<- FACTORIAL :: 1
     <<- FACTORIAL :: 1
     <<- FACTORIAL :: 2
     <<- FACTORIAL :: 6
     <<- FACTORIAL :: 24
     <<- FACTORIAL :: 120

  120
```

Notice how each value of FACTORIAL is passed back through every level of the recursive call, where it is multiplied by the value of n at that level. Work through the example to be sure you understand it.

Exercises

1. Write a recursive function that adds up the numbers in a list, for example (1 2 3 4) = 10.
2. Write a recursive function that takes a list and returns everything minus the last element.

Answers

1.
```
(DEFINE (SUM (LAMBDA (L)
   (COND
      ((NULL L) 0)
      (T (ADD (CAR L) (SUM (CDR L))))
   )
)))
```

2.
```
(DEFINE (RDC (LAMBDA (L)
   (COND
    ((NULL (CDR L)) NIL)
    (T (CONS (CAR L) (RDC (CDR L))))
   )
)))
```

Contents | The Conditional | The Lisp Editor ED

LEARNING LISP

Contents | Simple Recursion | Lists as Trees

The Lisp Editor ED

In the previous chapters, you were at the mercy of your typing ability. If you could type your Lisp function in correctly the first time, then more power to you. However, for the rest of us, even if we do manage to get the parentheses balanced, our functions usually don't work perfectly from the outset. Provided with Lisp is a mechanism for manipulating user defined functions. It is called ED, and it is designed to edit functions.

You get into the editor by typing

```
:(ed fun)
```

where FUN is the name of the function you want to edit. The editor will respond with a level indicator and a *point of view* [POV]. This will typically look something like this:

```
TOP: (LAMBDA & &)
```

The point of view is a window into the function. That is, you are always looking [through the eyes of the editor] at some particular part of the function body and the point of view is a picture of that part of the function.

Let's define a little function in order to play with the editor. Here is the function EDITME:

```
:(DEFINE (EDITME (LAMBDA (L)

:   (PRINT (CAR L)))))))

  EDITME
```

Here is what happens when we try and edit the function.

```
:(ED EDITME)
  TOP: (LAMBDA & & )
```

The editor responds by displaying the top level of the LAMBDA list. Any atoms [in this case only

LAMBDA] are fully spelled out, but any lists are represented by a &. The reason for this is that many lists are quite large; you would not want to see the entire list, only enough to give you an idea of where you are.

The rule to remember in editing parts of lists is, "what you see is what you edit". Whenever you do anything in the editor, you are shown the current expression [the POV]. If you want to see the entire POV without the &, type a P. This will invoke the pretty-printer on the current POV. The pretty-printer shows you what the entire window looks like and tries to indent its elements in some reasonable manner.

```
?P

  ( LAMBDA (L )
    (PRINT (CAR L ) )
  )

  TOP: (LAMBDA & & )
```

We can change our location in the list [that is, change the POV so that it indicates a different part of the list] by entering a number as the editor command. The number is the position in the current window to which you would like to move. Lists are numbered like this:

```
(LAMBDA (L) (PRINT (CAR L)))
------- --- ----------------
   1     2         3
```

If we want to make the first element the new current expression we can simply type

```
?2
  TOP:2: (L )
```

Use "0" to move back one level.

```
?0
  TOP: (LAMBDA & & )
```

We cannot go to 1, because the window isn't allowed to be just a lone atom. You can only go to a list, which shows up in the window as an &. Notice that the numbering changes once we move to a new part of the list.

```
?-1
   TOP:3:  (PRINT &)
```

Negative numbers can be used to move down into the function, but the elements are counted from the right end. Therefore, we can number the current expression as follows:

```
(PRINT  (CAR L))
------  --------
  -2       -1

?-1
   TOP:3:2:  (CAR L)
?-3
   INVALID ELEMENT.
```

We cannot go to something that doesn't exist, and the editor will report this to you. Let's go back to the top.

```
?0
   TOP:3:  (PRINT &)
?0
   TOP:  (LAMBDA & &)
?0
   NO UP FROM HERE.
```

If you also try to go past the top, of course the editor stops you.

There are also provisions for moving BACK one element and to the NEXT element. These commands are BX and NX respectively. The last movement command is GO (LEVEL INDICATOR). Regardless of where you are, this will move you to a completely new place. Make sure the argument to GO is a list. Let's see how these work.

```
?2
   TOP:2:  (L)
?NX
   TOP:3:  (PRINT &)
?GO(2)
   TOP:2:  (L)
```

Something funny will happen if the NX or BK are asked to go to atoms. The editor won't let an atom be the only thing in the window, but by jumping NX or BK you might be asking it to make an atom the current window. Well, the editor is smarter than that. It will skip over atoms that are in the way when BK or NX are done.

So far all we've done is move around within the function. Now, let's try and change something. To do

this, we need to use the "insert" command. Remember that editing commands only modify the window. Make sure that what you want to modify is there. Move around until you are looking at the correct section of the function.

```
?0
  TOP: (LAMBDA & & )

?I NOT AFTER 2
  TOP: (LAMBDA & NOT & )

?I ONLY BEFORE -1
  TOP: (LAMBDA & NOT ONLY & )

?I WAS FOR 1
  TOP: (WAS & NOT ONLY & )
```

Note the three different forms of the insert ["I"] command. You can insert AFTER an element, BEFORE an element, or FOR an element. These three can all be abbreviated to a single letter each [A, B, or F]. You are also not restricted to inserting just atoms either. You can insert full lists.

```
?I (NOT) FOR 3
  TOP: (WAS & & ONLY & )
```

There is also a delete command for removing items from a list.

```
'D -2
  TOP: (WAS & & & )
```

Again, for the delete command, you can use either positive or negative numbers.

Using the insert and delete commands we can perform any list surgery we want. Feeling very confident in our abilities, let's try to fix up a real function. First, we must leve the editor by typing ABORT. ABORT will completely ignore any changes we have made, but since we just made a wreck of EDITME, that's perfectly acceptable.

```
?ABORT

  EDIT ABORTED.
```

Now, we must enter the definition of the function. It may already be there, but you should re-enter it anyway.

```
:(DEFINE (ENDS (LAMBDA (L)

:   (CONC (FIRST L) (LAST L)))))))

  ENDS
```

Now, let's go back into the editor.

```
:(ED ENDS)
  TOP: (LAMBDA & & )
```

We have loaded the function into the editor, so let's get started by looking at the whole function.

```
?P

  ( LAMBDA (L )
    (CONC (FIRST L ) (LAST L ) )
  )

  TOP: (LAMBDA & & )
```

Great. Let's move down the function.

```
?3
  TOP:3: (CONC & & )
```

Now, remember what we have to do. We have to make (FIRST L) the new list (MAKELIST (FIRST L)), and the list (LAST L) the new list (MAKELIST (LAST L)). In English, we have to make a list out of an atom.

```
?I (MAKELIST (FIRST L) ) FOR 2
  TOP:3: (CONC & & )
```

OH NO! No visible change! Did it work? Why does it still say CONC & &? Yes, it worked. We can use the pretty-printer to verify this.

```
?P

  (CONC (MAKELIST (FIRST L ) ) (LAST L ) )


  TOP:3: (CONC & & )
```

Okay. We still have one more to do, so let's do it.

```
?I (MAKELIST (LAST L)) F -1
  TOP:3: (CONC & & )
```

Note the use of the F for the "for" option. As we said, all three options can be abbreviated to just the first letter. Now, all that we have to do is leave the editor and save the modified function. EXIT will do both for us.

```
?EXIT

  NIL
```

Not that we don't trust our work, but why don't we test the function just to be sure.

```
:(ENDS '(AUNT EDNA IS A PICKY EATER)))

  (AUNT EATER )
:
:(ENDS '(SO YOU THINK YOU ARE FUNNY)))

  (SO FUNNY )
```

Looks like it worked. That sumps up the editor commands. Don't forget that the editor is written in Lisp, so you should feel free to look at it and play with it. The code for the whole editor is shown in an appendix to this text. Sometimes it's useful to just look at a function without having to go all the way into the editor. You can get to the pretty printer from outside ED by using the PPRINT function. Just say

```
:(pprint fun)
```

where "fun" is the name of the function. It will display the function on your screen in a neat form.

THE EDITOR AS A LISP PROGRAM

In the appendix we have included the entire text of the Lisp editor. It is all written in Lisp. After reading through the rest of the book, you should definitely try to understand how the editor functions. This will probably be the most valuable exercise that we offer in this book. The editor is a very complicated Lisp system and although it certainly isn't an artificial intelligence application of Lisp, it is a very useful one.

Contents | Simple Recursion | Lists as Trees

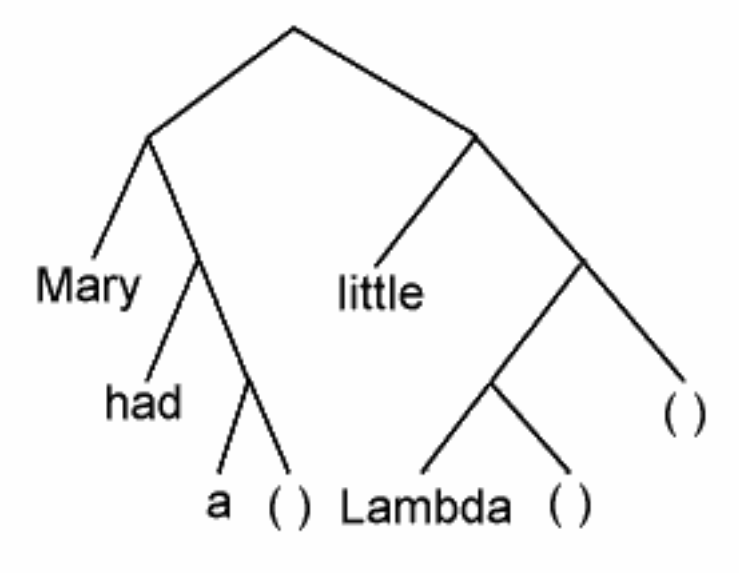
LEARNING LISP

Contents | The Lisp Editor ED | Trees and Recursion

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 useful way to see 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

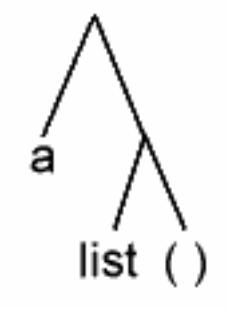
```
((Mary had a) (little (lambda)))
```

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 the two way branching.

Let's take a look at a simpler tree.

```
(a list)
```

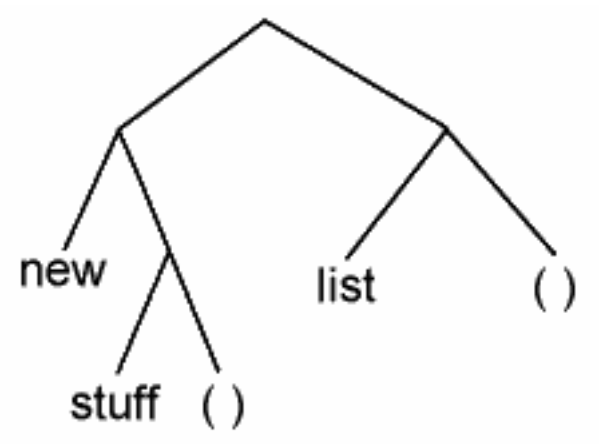


This tree has two nodes and four branches. There are atoms at the ends of the lowest branches [NIL is an atom too!].

```
(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 the following diagram.

```
((new stuff) list)
```

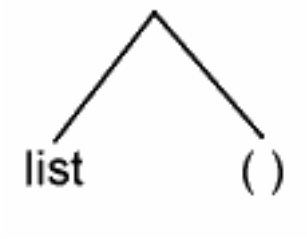


When we look at the CAR of the above tree we are looking at

```
(new stuff)
```

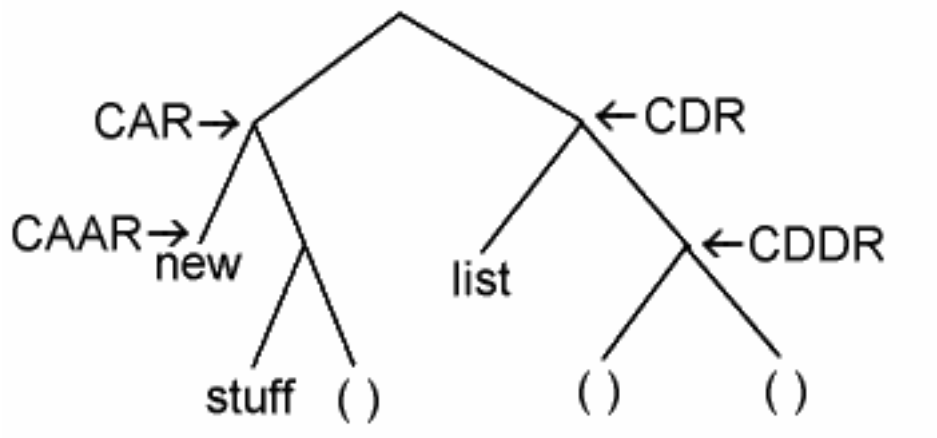
CAR can be thought of as returning all the material hanging on the left branch. CDR, as you might expect, returns the material on the right branch.

```
(list)
```



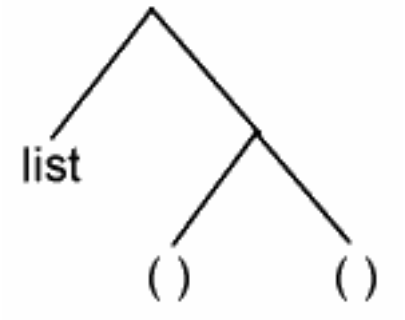
This is why there are still a set of parentheses around it. Note that all the trees eventually terminate in NIL ["()"]. Recall that if you keep taking the CDR of a list you will eventually hit a NIL. The reason for this should now be apparent. If we were to insert a NIL into the end of the list, we would get:

```
((new stuff) list ())
```



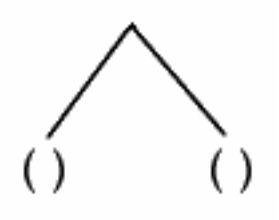
This is the CDR of the above list.

```
(list ())
```



The CDR of that is

```
(())
```



CAR of that is

```
()   [that is: NIL]
```

NIL

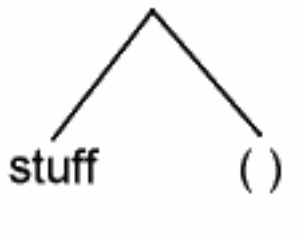
So is CDR.

Both the CAR and CDR of NIL are NIL. Thus, we can't change the list from here on without CONSing stuff onto it. Let's do so now.

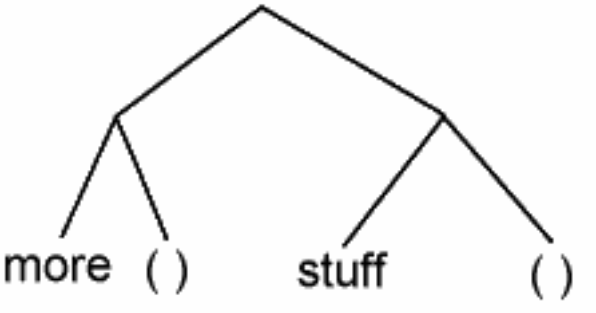
```
(setq worktree ())
```

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.

Contents | The Lisp Editor ED | Trees and Recursion

LEARNING LISP

Contents | Lists as Trees | A Style of Programming

Trees and Recursion

We promised you that trees would help you understand recursion and it will. Most functions in Lisp, whether they are built-in or written by us, 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 removed the far left 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 trees themselves, then they would simply be printed out as-is:

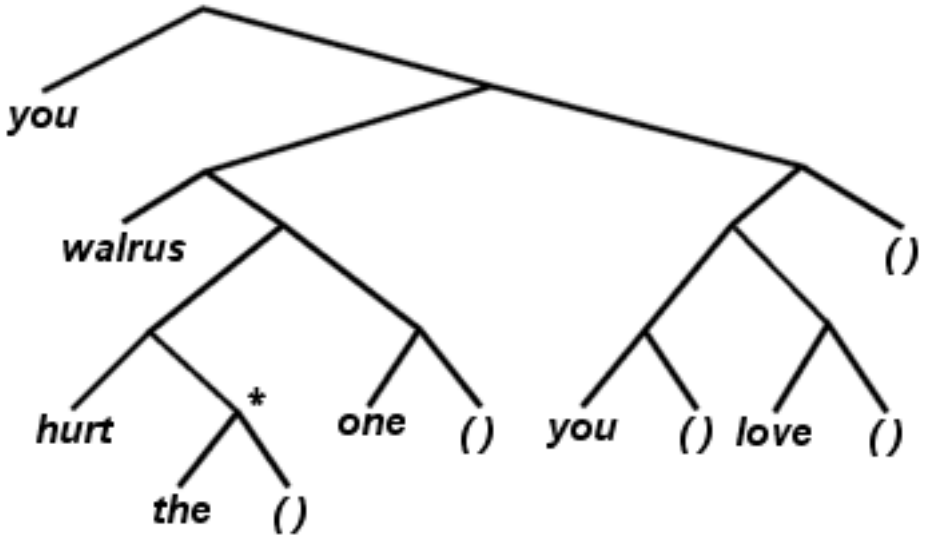
```
:(DEFINE (RECITE (LAMBDA (STUFF)

:    (COND

:       ((NULL STUFF) ())

:       (T (PRINT (CAR STUFF))

:          (RECITE (CDR STUFF)))))))


:(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 redefine RECITE with a COND as suggested by the previous description. We keep calling RECITE of the CAR [and the 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?

```
:(DEFINE (RECITE (LAMBDA (S)

:  (COND ((ATOM S) (PRINT S))

:        (T (RECITE (CAR S))

:           (RECITE (CDR S)))))))

  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 print "the" [as indicated by the star ["*"] in the above picture], 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 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 smaller and smaller trees to the later calls, we can scan the entire main tree.

Contents | Lists as Trees | A Style of Programming

LEARNING LISP

Contents | Trees and Recursion | Scope Considerations

A Style of Programming

The result of a simple recursive function is that the value returned is the value of the last evaluated expression which does not involve a recursive call. Typically, this is a T or NIL or some atomic value. Since the function is stopped 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 complex recursion.

Suspended evaluation is of primary importance here. When a function is evaluating its arguments, it is suspended until all those arguments are through 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 may involve another occurrence of that expression. The only way to understand what we're driving at is to see it happen. TRACE is the fastest way to do this and 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." This function is similar 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 handles 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 nor an atom. It flattens the CAR of L, flattens the CDR of L, and then CONCs the two results together.

There is a certain style to these recursive functions, and now is the time to expand on this. We always use the same format. First handle the termination conditions, then deal with special cases, and finally, 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))))))))

:     (T (CONC (EVIL-FLATTEN (CAR L))

:           (EVIL-FLATTEN (CDR L)))))))

  EVIL-FLATTEN
```

```
:(EVIL-FLATTEN ' (EINS ZWEI DREI))

  (EINS ZWEI DREI NIL )

:(EVIL-FLATTEN '(OOPS (BLOOPS) HOOPS))

  (OOPS BLOOPS NIL HOOPS NIL )
```

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

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 (----------)) ) )))
```

The hyphens 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 put the CAR back on the right end of this expression then we have the reverse function.

```
:(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 ) )
```

While an "assume it works" strategy may seem a little bizarre, it is a very useful and quite valid method. 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 return the NIL. This isn't quite what we had in mind. What we want to do is to stop recurring just before we get to the end of the list. Try this termination phrase: "((null (cdr L)) (car L))". This will 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 '(BIRD (NEST))))

  (NEST )
```

Exercises

1. Write a recursive function to perform multiplication according to the following formula:

   ```
   NxM=Nx(M-1)+N; Nx1=N;
   ```

2. Write a recursive function to decide whether a list is palindromic. A palindromic list is one which reads the same backward or forward. Assume that NIL and a list with just one element are palindromic. Also, you may want to use a help function or two.

Answers

1.
```
(DEFINE (FN (LAMBDA (N M)
  (COND
    ((EQUAL M 1) N)
    (T (ADD N (FN N (SUB M 1))))
  )
)))
```

   Note that this function assumes M is always greater than or equal to 1; if M is less than 1 the function will recur forever.
2. We can use the RDC function defined in chapter 11 (exercise #2, Answer) and the RAC function defined in chapter 15.

```
(DEFINE (PALEN (LAMBDA (L)
   (COND
     ((NULL L) T)
     ((EQUAL (CAR L) (RAC L)) (PALEN (CDR (RDC L))))
     (T NIL)
   )
)))
```

Contents | Trees and Recursion | Scope Considerations

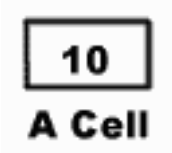
LEARNING LISP

Contents | A Style of Programming | Maps

Scope Considerations

Lisp functions live, work, and play in an environment. We learned how to save that contents of the environment a while back. 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 predefined variables, T and NIL. Using SETQ to define your own variables causes Lisp to set up an area of storage for holding 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 A.

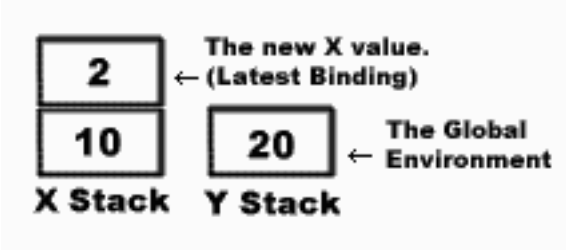
The interaction of the assignment with the evaluation of user-defined functions is interesting. Remember that the global value of variables are left untouched even though the local variables in the function may have the same name. Let's define some functions with which we can experiment.

```
:(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 Z))))

  FUN3

:(FUN1 2)
  2
  20
  2
  20
  3
  2
  5

:X

  10
```

```
:Y

  5
```

We defined three somewhat contrived functions. We begin 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 esists. If there is such a conflict, the new value is stacked on top of the older value or values. This can be better understood with the help of the following picture:



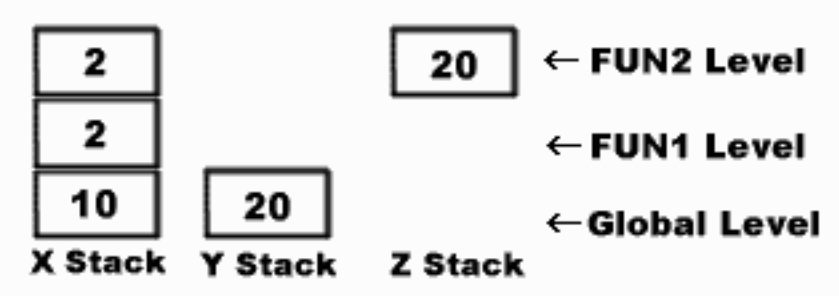
The value of the atom in evaluation is designated by the value at the top of its stack. Therefore, the value of X at this point is 2 and the value of Y is 20. These values are called a local environment.

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.

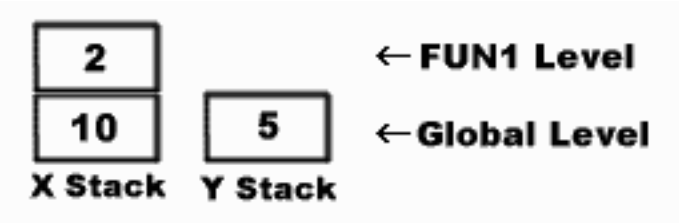
Back to our example. We are now inside the function FUN1. The first expression to be evaluated is "(PRINT X)". This causes the current value of X to be printed. Since X is a local variable to this function, the current value of X is the value of the argument supplied in the call of the function FUN1. This is the number 2 which we typed ourselves. Therefore, it is the first 2 in the output. The next expression, "(PRINT Y)", prints the global value of Y, which is 20.

The next expression, "(FUN2 X Y)" is a call to the function FUN2. The formal arguments for FUN2 are X and Z. Because X already exists in the environment [twice, in fact], we must again "stack" the new value, 2, on top of the older value, 2. [Note that the old value and the new value are the same. Lisp does not care, it will save the old one anyway.] Also, the value of 20 is bound to Z. The new environment looks like this:

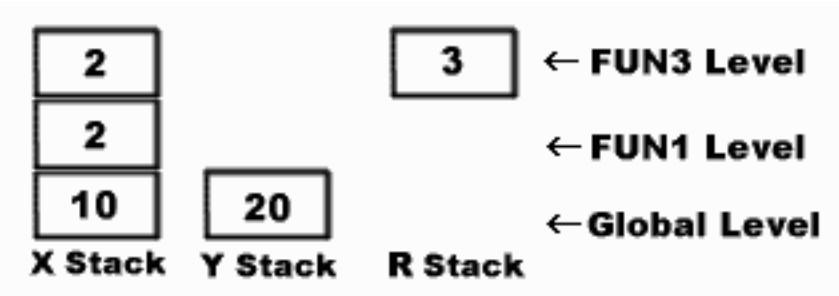


We are now inside of FUN2 [which is inside of FUN1]. The value of X is printed as 2, in the third output line. Then, the value of Y is printed. This is 20. Next, the value of Z in printed [20, the fifth line]. We then use the SETQ function to assign a value to the variable Y. Therefore, Y now has the value of 5. If Y had been a local variable in FUN2, then the assignment would have been broken when leaving FUN2. However, since Y is global, this assignment of value is permanent, at least until the next assignment. The action of changing the value of a variable which is not one of the function's formal arguments 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 uppermost level has been removed from the stack.



Next, we 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 is FUN1. There, X had the value 2. Therefore, a 2 is printed for X. What about Y? 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 last number printed is the value returned from FUN1. FUN1 is returned from FUN3, which is returned from PRINT. Remember that everything in Lisp returns a value, even PRINT.

Contents | A Style of Programming | Maps

LEARNING LISP

Contents | Scope Considerations | Isplay Ogrammingpray

Maps

We are going to make life with Lisp much simpler. The careful reader may have noticed some patterns in the recursive Lisp functions that we have been defining. Lisp provides built-in functions to perform a number of these patterns.

Suppose that we want to perform the same 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. INT is a new Lisp function that returns the value of its argument preceding the decimal point, if any. For example, (INT 33.89) is 33. Now for EVEN. It is a predicate that works by dividing the argument by two and determining whether that result is the same as the integer part of the number divided by 2. Here's EVEN:

```
:(DEFINE (EVEN (LAMBDA (N)

: (EQUAL (DIV N 2) (INT (DIV N 2)))))))

  EVEN
```

Now we can use MAPCAR to apply EVEN to all the members of a list.

```
:(MAPCAR 'EVEN '(0 1 2 3 4 ))
```

```
  (T NIL T NIL T )

:(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: As its arguments MAPCAR takes 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)". 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 the elements of each sublist but left the order of sublists intact.

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

```
:(DEFINE (EVENMAP (LAMBDA (L)

:  (COND

:  (( NULL L) NIL)

:   (T (CONS (EVEN (CAR L))

:   (EVENMAP (CDR L)))))))
  EVENMAP
:(EVENMAP '(0 1 2 3 4 5 6 7 8)
:))

  (T NIL T NIL T NIL T NIL T )
```

Contents | Scope Considerations | Isplay Ogrammingpray

LEARNING LISP

Contents | Maps | FEXPRS: Unevaluating Functions

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. These steps are, of course, dependent upon the exact problem specified, but there are some general rules which we will emphasize.

The problems we will attack are those of a pig Latin translator. The system [a collection of functions and help functions all intented to solve one problem] will take a sentence in English and return the sentence in pig Latin. It will do this:

```
:(PIGLATIN '(TAKE OUT THE TRASH))

  (AKETAY OUTWAY ETHAY ASHTRAY )
```

The first step in designing a system is: Take out a piece of paper and a pen or pencil.

You will need paper to keep notes [and doodle while you think]. If the system will have many functions in it you're definitely going to need a scorecard to keep track of the functions and their arguments. Because of the environment nature of Lisp, it's not easy to comment the functions so notes are important!

Next, define the problem and lay out exact specifications.

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

The next step is to decide on an algorithm.

The standard pig Latin algorithm works in the following manner.

Look at each word in the sentence. If its first letter is a vowel, simply add "way" to it. Otherwise, remove all the letters from the beginning of the word, and up to the first vowel. Put them after the word. Then add "ay".

There are a few assumptions in this description. It is very important to specify your assumptions so that

you know which cases you won't have to deal with. A well-defined set of assumptions can make a programming task quite a bit simpler.

We have assumed that all words have vowels in them. We will ignore the case of "words" (actually syllables) with no vowels. Also, we assume that only vowels can be one-letter words ("a", "I"). By examining some sample cases, we can fill in blanks in the algorithms.

```
WATER --> ATERWAY
WEEPS --> EEPSWAY
```

Thus, "W" is not a vowel.

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

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

We've now 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. The first part of the algorithm says: "Look at each word in the sentence." Because our sentence is a list, the words are the atoms in that list. Looking at each word is similar to the MAPCAR operator. 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. The first is its name [and the name of the help function it calls] have meaning. We could have called this function "xyzzy" but then we would never be able to remember what it did!

The second point is more subtle. The main function 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 manageable increments. There are many advantages to this, not the least of which is that simpler and smaller functions are easier to edit or retype.

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 [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 a list and then turn the processed list back into an atom:

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

Of course, Lisp conveniently provides functions for doing this. EXPLODE turns an atom into a list of its letters, and IMPLODE does the opposite.

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 implosion. We will define PIGWORD as follows:

```
:(DEFINE (PIGWORD (LAMBDA (WORD)

:    (IMPLODE (PIGLISTEDWORD (EXPLODE WORD)))

:    )))

  PIGWORD
```

PIGLATIN calls PIGWORD via MAPCAR and passes it to each word. PIGWORD, in turn, explodes the word and processes it using another help function [with a meaningful name] called PIGLISTEDWORD. PIGLISTEDWORD will return the list of the "pigized" word, and PIGWORD will implode it to an atom again and return the new word to PIGLATIN.

All that remains now is write PIGLISTEDWORD. 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)))))

  PIGLISTEDWORD
```

Several things are still missing from the system. PIGVOWEL will translate a word that starts with a vowel into its pig Latin equivalent. PIGNOVOWEL will translate all other words, and the predicate ISAVOWEL will tell us whether the letter it received is a vowel or not.

Let's write the simple 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 the simple part of the system--sentences where all the words begin with a vowel.

```
:(PIGLATIN '(ALERT AIRMEN ARE ONTIME))

  (ALERTWAY AIRMENWAY AREWAY ONTIMEWAY )
```

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

Great! Most of our system finished in only 5 functions. All we have to do now is write the hard one. PIGNOVOWEL. Let's describe the responsibility of that function in detail.

Since we can only deal with one letter of the exploded word at a time, we have to search for the first 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 the information as we go? The answer is to 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, and 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)))))))

  PIGNOVOWEL
```

Note that there are two arguments 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) ())

  (ISTHAY)
```

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

```
:(PIGLATIN '(EVERY GOOD BOY DOES FINE))

  ** ERROR: TOO FEW ARGS **
  PIGNOVOWEL :: (WORD)

+()

  NIL
```

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

```
:(DEFINE (PIGLISTEDWORD (LAMBDA (WORD)

:  (COND

:    ((ISAVOWEL (CAR WORD)) (PIGNOVOWEL WORD))
```

```
:    (T (PIGNOVOWEL WORD NIL))))))

  PIGLISTEDWORD


:(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:

- Take notes. Write down every function and its "form of call". The form of call for ADD, for example, is: "(ADD number1 number2)". Also, write one line describing what is does and what it comes back with. If you're brave you can write down the function definition. This will help you figure out what's wrong.
- Formulate a complete specification of the problem before beginning to even think in Lisp. The I/O behavior of the function should be the main part of this description. I/O behavior also is a good way to formulate the descriptions of most help functions.
- Define the algorithm and assumptions [restrictions] involved. This will save you from embarrassing trouble later, when you suddenly remember something that you had forgotten.
- Use top down programming. That is, start with the function that the user will type in (think of the user as someone other than yourself), then fill in your undefined functions as you get to them. Work your way deeper and deeper into the 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 brief. Short functions are easier to edit and easier to retype.
- Implement a single part of the algorithm in each function. This will also help keep them brief. If the algorithm is well described, then there should be approximately 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.]

Designing a system in Lisp or any other programming language, is like designing in general. You take it step by step and fill in the unknowns when you come across them. Always have the final

goal in mind.

Exercises to Translate Into Thoughts

1. Explain how a function that sorts words in a list would work. Think about each function and describe the entire system in English.
2. Practice taking a paper and pencil [or pen] out before starting to write things into Lisp. Do this until it becomes automatic.
3. There is a simplification that can be made in our Pig Latin system. It is possible to do it without a function because its action is performed by a special use of another function. What are we talking about? Fix the system accordingly. Think about the advantages and, especially, the disadvantages of doing things this way.
4. How is top down programming like recursive tree traversal? Design a Lisp system [in English but with possible real help functions] that would take a problem specification. Write the program using top down tree traversal.
5. Adjust the pig Latin translator to work with words such as, "nth", "cwm" and "crwth".

Answers

3. PIGVOWEL is a special case of PIGNOVOWEL. We can redefine PIGLISTEDWORD as

```
(DEFINE (PIGLISTEDWORD (LAMBDA (WORD)
  (COND
   ((ISAVOWEL (CAR WORD))
      (PIGNOVOWEL WORD '(W)))
   (T (PIGNOVOWEL WORD '()))
  )
)))
```

5. Put a termination condition in PIGNOVOWEL to watch for vowelless words.

```
(DEFINE (PIGNOVOWEL (LAMBDA (WORD LETS)
  (COND
    ((NULL WORD) (CONC LETS '(A Y)))
    ((ISAVOWEL (CAR WORD))
      (CONC WORD (CONC LETS '(A Y))) )
    (T (PIGNOVOWEL (CDR WORD)
      (APPEND (LETS (CAR WORD))) )
  )
)))
```

Contents | Maps | FEXPRS: Unevaluating Functions

LEARNING LISP

Contents | Isplay Ogrammingpray | Control Structures

FEXPRS: Unevaluating Functions

Most of the functions that we've used thus far are called "EXPRs" [short for EXPRession]. An EXPR is a function that evaluates its arguments when it is called. ADD, MULT, 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 as is to the function. A function that does not evaluate its arguments is called a "FEXPR".

The first flexibility that this offers us is that we can type things in without quotes. The pig Latin example could have been simplified with a FEXPR. 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. There is exactly one formal argument, never more, never less:

```
:(DEFINE (PRINTME (FLAMBDA (INPUT)

:         INPUT))))

  PRINTME
```

We have defined a trivial function in order to 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 FOR YOU)

  (I HAVE BUT ONE LIST FOR YOU )
```

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

argument.

```
:(PRINTME (CAR (BUS)) (CDR (CADDR CDDAR)))

  ((CAR (BUS ) ) (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.

Now let's do something less trivial. Notice that PRINTME always puts an extra set of parentheses around its argument.

```
:(PRINTME LIBERTY)

  (LIBERTY )

:(PRINTME (OR DEATH))

  (OR DEATH ) )
```

Here's a version of PRINTME that doesn't do that.

```
:(DEFINE (PRINTME2 (FLAMBDA (INPUT)

:   (CAR INPUT))))))

  PRINTME2
```

Now PRINTME2 acts exactly like the QUOTE function that we have encountered in our Lisp studies. In fact, this is exactly what the quote (') 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 ) )

:(PRINTME2 (CAR (DONT EVEN TRY)))

  (CAR (DONT EVEN TRY ) )

:(QUOTE (CAR (DONT EVALUATE THIS)))
```

```
  (CAR (DONT EVALUATE THIS ) )
```

Knowing this might be of some relief to those of you who have noticed that when an error occurs, Lisp has expanded the apostrophes into the word QUOTE.

So, there has been at least one FEXPR with us since the beginning. Can you think of others? How about SETQ? Why doesn't the first argument in SETQ need a quote? [Why do you think that they call it SETQ?!] Wherever it seems as though something should be quoted but is actually not necessary, there's probably a FEXPR in the works someplace.

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 the list of input elements. Using this rule we can rewrite PIGLATIN as,

```
:(DEFINE (PIGLATIN (FLAMBDA (SENT)

:   (MAPCAR 'PIGWORD SENT))))))

  PIGLATIN
```

You should try this and verify 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 paired names and phone numbers, and returns each pair in list form.

```
:(DEFINE (PAIRWISE (FLAMBDA (IN)

:    (SEGMENT IN))))))))

  PAIRWISE

:(DEFINE (SEGMENT (LAMBDA (L)

: (COND ((NULL L) ())

:  (T (CONS (LIST (CAR L) (CAR (CDR L)))

:    (SEGMENT (CDR (CDR L)))))))))

  SEGMENT
```

```
:(PAIRWISE A 1 B 2 C 3 D 4)

  ((A 1 ) (B 2 ) (C 3 ) (D 4 ) )
```

There is another important rule of FEXPR's: 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 (FLAMBDA (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.

It would be nice to be able to TRACE FEXPRs but P-Lisp can't. Other versions of Lisp may or may not allow this. However, for this example, we'll imagine that it can and imagine that it will print.

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

The first list went in okay, but it looks like the recursive steps didn't work correctly. Since the FEXPR doesn't evaluate its arguments, the "(cdr L)" wasn't evaluated, and the next iteration simply tried to do REVLIST on "(cdr L)" as opposed to the CDR of "L". This would have gone on forever if we hadn't

interrupted. Actually, Lisp has a large but finite recursion limit. You will undoubtedly encounter RECURSION CHECK errors eventually. That's what really happens if you let a recursive function run away.

Exercises Not to Be Evaluated

1. Write REVLIST correctly. You'll probably want to use a help function.
2. Is DEFINE a FEXPR? Of course, it is, otherwise we wouldn't have asked the question. Convince yourself with the argument just presented.

Contents | Isplay Ogrammingpray | Control Structures

LEARNING LISP

Contents | FEXPRS: Unevaluating Functions | Eval and Apply

Control Structures

We claimed that Lisp was a programming language, and programming languages usually do complicated things. We did one fairly complicated operation but, hopefully, it wasn't too difficult (the pig Latin system). In order to do even more complicated work in Lisp, we need to be able to do some of the things that other languages do.

Control Structures are something that most programming languages provide in order to help the programmer organize his or her thoughts and, thus, lend better organization to the program. They can be thought of as a frame into which you can shape your program. For example, we have been shaping our programs so far into the frame of CONDs and recursion. These are one type of control structure.

This chapter will show you how certain useful control structures are implemented in Lisp. Programming with these structures is a matter of experience. You'll eventually learn structures that are not recursive, and then we will return to this chapter and point out alternate methods of programming.

One thing that other languages do which we've seen a lot of in Lisp is the COND. In Pascal, COND can be thought of as a CASE statement or a series of IF-THEN-ELSE clauses. Here is a COND written in a language called *pseudo code*.

```
:(DEFINE (MEMBER (LAMBDA (A L)
   IF L is NULL THEN
     NIL
   ELSE IF (CAR L) equals A THEN
     T
   ELSE
     (MEMBER A (CDR L)))))
```

That's our old friend MEMBER. You might try writing MEMBER in your favorite language. (Use a list of numbers instead of atoms--that will make it simpler.) You'll find that your tendency in language other than Lisp is to write a loop. Here is MEMBER written as a loop.

```
:(DEFINE (MEMBER (LAMBDA (A L)

:  (PROG ()
```

```
:     TOP

:     (COND

:        ((NULL L)  (RETURN NIL))

:        ((EQUAL (CAR L) A)  (RETURN T)))

:     (SETQ L (CDR L))

:     (GO TOP)))))

  MEMBER
```

What we have here is a special sort of control structure called a PROG. That is, obviously, short for "program". PROGs look like this.

```
(PROG (locals) obj1 obj2 obj3 . . .)
```

Identify each part in the PROG above. Note that there are no locals and that obj1 is the atom TOP. The other objects are lists that have Lisp expressions in them. OBJ4 has an unusual function called GO in it. We'll explain all this now.

PROG works like this: Before it begins it sets all the locals to NIL. The old values of these locals are saved, and these new values [the local ones] are used only while Lisp is doing the PROG [Remember the scope chapter]. Thus, when the PROG is done, whatever old values were in the locals come back.

PROG begins with the first object. If it is an atom [like TOP], then it simply ignores it. Why? You'll see. If the object is not an atom then it evaluates it and if it can, goes on to the next object. That's really all there is to a PROG! Simple!

We haven't answered some questions yet: What's TOP for? What does GO do? How does PROG return a value from its calculation? The answer to this last question will be obvious if you carefully study the COND in our PROG example. There is a special function called RETURN. The argument to RETURN is returned from the PROG that it is in, and that PROG stops!

GO and the atom TOP are used to implement looping. The reason that PROG evaluation ignores labels [which are just atoms by themselves] is that they are simply markers to name various places in the PROG. They are *labels!* TOP is a label that marks the top of the program [the place that we want to go back to each time]. The names of labels are arbitrary and there can be many in one PROG. When PROG evaluation encounters a GO expression, it hunts around in the PROG body for an atom object that has the same name as the argument of GO. Evaluation begins again there.

This is quite simple but it's not clear what you would use it for. Let's do a complicated PROG example: Let's rewrite the pig Latin system as one giant PROG. That is, we're going to incorporate all of those help functions into the body of a single function.

Before we show you the function, let's talk about comments a little bit. Comments are *very* important in programming. Usually, when you write a program, you put in "comment lines" to tell others what's going on in various sections of the program, or to remind yourself. There are two reasons why we haven't discussed them yet. First, all the Lisp programs that we've written thus far have been very small. So small, in fact, that they *should* have been commented merely by virtue of having a good meaningful name!

The second reason why we've avoided comments is rather poor. That is, that Lisp does not handle comments very well. You can get the Lisp language to simply ignore a line that you type, by preceding it with a semi-colon.

```
:;THIS IS A COMMENT

:(CAR '(A

:; THIS IS A COMMENT TOO!

:B)))

  A
```

This is fine and lets us comment as we are entering things but exactly because Lisp literally ignores these lines, they don't stay around with the program. Therefore, they aren't very useful. If you enter a program and put in comment lines, they won't appear on the printed display because they have been ignored *on entry!*

You may think it would be easy to make a COMMENT function that lets us put comments into the functions. Here's a possible function to do just that.

```
:(DEFINE  (COMMENT (FLAMBDA (S) ())))))

  COMMENT
```

This function simply eats its arguments and returns (). This is similar to ignoring the comment. However, this won't work very well.

```
:(COMMENT THIS IS A COMMENT)
```

```
  NIL

:(PROG () (PRINT 'TESTING)

:      (COMMENT THIS IS A COMMENT)

:      (PRINT '(1 2 3)))))
  TESTING
  (1 2 3 )

  NIL

:(COND ((NULL 5) (PRINT 'NOPE))

:    (COMMENT THIS IS A COMMENT)

:    (PRINT '(1 1 1)))))

    ** ERROR: UNDEFINED ATOM **
      EVAL :: COMMENT

+()

  NIL
```

What happened? It looks like we can put comments into PROGs easily but only in specific places. Unfortunately, there's no simple solution to this problem, and the more complex you make it, the harder it gets. Therefore, because this particular function is so large, we will use comments, but if you type them in as we show you here, don't be surprised when they disappear!

But back to the point. Here's the pig Latin function as a PROG.

```
(define (piglatin (lambda (s)
;
; The locals to the prog are:
; result - will store the piglatin form as it is made.
; word - holds the word that is being translated.
; newword - gets the result of a translation from word.
; holdchars - sabes up consonants while the word is being scanned.
;
(prog (result word newword holdchars)
;
```

```
; Come back here to see if we're done. If not then get the
; next word from s and put it into word. Then set up everything
; for a single word translation. Note that '(w) gets inserted
; into the character list if the word starts out with a vowel.
;
  nextword
     (cond ((null s) (return result)))
     (setq word (explode (car s)))
; Remove the word from the front of s.
     (setq s (cdr s))
     (setq holdchars '(w))
     (setq newword ())
;
; This loop translates the word in word. Each character is
; checked for vowelness (we actually still use ISAVOWEL here).
;
  wordloop
     (cond
      ((null word)
       (setq result
             (conc result
              (list (implode (conc newword
                                   (conc holdchars '(a y))
                                     )))
             ))
       (go nextword) ; This word is done so go get a new one.
     ) ; This paren closes this condition of the COND.
;
; If the letter in front is a vowel then move the whole word
; to newword and arrange for the loop to stop by killing word
; to NIL.
;
      ((isavowel (car word))
       (setq newword word)
       (setq word ())
      )
;
; The following two cases take care of the things to do when
; there is a consonant in front of the word. If [w] is
; still in holdchars we've got to kill it. Otherwise simply
; put the letter from the front of the word into the holding
; set and remove it from the word, then go on.
;
```

```
      ((equal '(w) holdchars)
       (setq holdchars (list (car word)))
       (setq word (cdr word))
      )
      (t (setq holdchars (conc holdchars (list (car word))))
         (setq word (cdr word))
       )
      ) ; Close the COND sequence
       (go wordloop)
 ) ; Close the PROG
))) ; Close off the whole function
```

Whew! That was a lot of writing! Hopefully, you found that entirely counter intuitive. Wasn't the recursive, modular definition much simpler? We, of course, didn't have to put all that in one function, but that way we got to show you a lot of PROG utilization and a few ways that comments can come in handy when writing big programs.

PROGs are pretty useful but there are a couple of other control structures that are simpler and sometimes equally useful. These are AND and OR. As their name implies, AND and OR work with true [T] and false [NIL] statements. Let's look at some examples.

```
:(AND T T)

  T

:(AND T ())

  NIL

:(AND () T T)

  NIL

:(OR () T () )

  T

:(OR () () () )

  NIL
```

AND and OR take any number of arguments [yes, they are FEXPRs]. If *any* of the arguments of AND is false, then AND returns (). This is like saying "If Bill and Lester and Dave go to school then TRUE."

Likewise, OR returns false only if *all* its arguments are NIL [if *any* of its arguments are T]. Look at the above examples and try some on the computer.

By the way, the arguments don't have to be NILs and Ts:

```
:

:(OR (GREATER 3 5) (EQUAL 2 2))

  T

:(AND (GREATER 5 3) (EQUAL '(YES) '(NO)))

  NIL
```

There can be any list of expressions in an AND or an OR.

This is really very useful in COND predicates. It allows you to put many tests in one COND line.

```
(COND ((ORD (NULL L) (EQUAL 1 (LENGTH L))) blahblahblah))
```

This does "blahblahblah" if either of the two conditions are true.

Why is this chapter on control structures? Well, AND and OR control the evaluation of their arguments in an odd way. In order to determine the result of an OR, all we have to do is evaluate until the first expression returns T. If there is even one true expression, then the result of the whole OR is T. Thus, OR only evaluates until it finds a T. So the following expression will *never* get to do the printing. It never gets past the second equal because it has enough information by then to figure out what the result of the OR will be!

```
(OR (EQUAL 3 4) (EQUAL 4 4) (PRINT 'BOO!))
```

This is how OR controls execution! It is very important to remember that in Lisp everything except a NIL means *true*. Therefore, it doesn't take just a T to stop OR. It will stop at any expression which returns anything other than NIL. In fact, when an OR stops short, the result it that value which caused it to stop.

```
:(OR (EQUAL 2 2) (SETQ Y 32))

  T
```

Y never gets set!

What about AND? Well, same game except that AND works the other way around. In order to figure out whether the result of the AND is going to be false, it goes until it hits the first occurrence of a false expression. Then it has enough to determine that the result is false!

AND is true until proven false. OR is false until proven true! These can be used for work while they're on trial.

Exercise Your PROGramming Facilities

1. Our PROG version of PIGLATIN also doesn't handle "nth" or "crwth". Try to alter it so that it does. Is it easier or harder to fix the PROG as compared with the modular, recursive system?
2. Suppose you tried to fix the problem of using the COMMENT function in CONDs. You might try to give COMMENT a value, since that's what Lisp seemed to think was missing. What value would you give it so that CONDs like this worked correctly? Why is that still not correct? (Think about OR and AND.)
3. Write your own versions of AND and OR as FEXPRs with PROGs.

Contents | FEXPRS: Unevaluating Functions | Eval and Apply

LEARNING LISP

Contents | Control Structures | Properties and Lambda Expressions

Eval and Apply

In 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 [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)". To repeat the last line in the above example use the APPLY function.

```
:(APPLY FUN X Y)

  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 (MULT x x)) 15)
```

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

```
:(SETQ P '(ADD (MULT 3 (MULT X X)) 15)))
```

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

:(EVAL P)

  42
```

Now we have the capability to form polynomials and then evaluate them. EVAL gives us a very handy way of making FEXPRs much more powerful. Suppose that we wanted to write an addition function that used many arguments, no just two. We want to be able to write "(add* 1 2 3 . . .)" and get back their sums. Here's a possible FEXPR to do that.

```
:(DEFINE (ADD* (FLAMBDA (L)

:  (ADD-SUB L))))

   ADD*

:(DEFINE (ADD-SUB (LAMBDA (L)

:  (COND

:    ((NULL L) 0)

:    (T (ADD (CAR L) (ADD-SUB (CDR L)))))))

   ADD-SUB
```

Convince yourself that this works for "(add* 1 2 3 4 5)". Now try using SETQ to set up some values and use them in ADD*.

```
:(SETQ SOME 5)

  5

:(SETQ MORE 6)

  6

:(SETQ VALUES 7)

  7

:(ADD* VALUES MORE SOME)
```

```
    ** ERROR: BAD NUMERIC ARG **
      ADD :: ((CAR L ) (ADD-SUB (CDR L ) )
      )

+()

  NIL
```

What happened? Well, when recursion stopped down in ADD-SUB, it returned a 0 which the next level tried to add to the then-car of the list, SOME. Well, SOME is not a number! ADD can't deal with it like that! SOME is not an atom--it has a value, but its name isn't that value [it isn't like numeric atoms in that respect]. How do we get its value from its name? Right, EVAL! Here's a new definition of ADD-SUB that works:

```
:(DEFINE (ADD-SUB (LAMBDA (L)

:  (COND ((NULL L) 0)

:  (T (ADD (EVAL (CAR L))

:     (ADD-SUB (CDR L)))))))

  ADD-SUB
```

Convince yourself! Trace EVAL and ADD-SUB and see why.

Exercises to Evaluate in Your Head

1. There is a way to write ADD* by changing ADD* itself instead of ADD-SUB. It also relies on EVAL but it does the evaluation before ADD-SUB ever gets called. Can you think of a way of doing this? If so, fix ADD*. If not, look up MAPCAR and recurse through this problem!
2. Write a function called DEFUN that permits us to get rid of some of the irritating parentheses. We want to be able to do this:

   ```
   :(defun function-name (args list) body . . .)
   :(defun function-name fexpr (arg) body . . .)
   ```

   Have it fill the material that DEFINE wants to see and then call DEFINE. Notice that unless we say "fexpr", it makes an EXPR. This will need a special test.
3. Write a function called NEWSETQ that counts the number of times it is used. It should look exactly like SETQ as far as its arguments are concerned. You should also keep the count in some

global variable called NEWSETQ-USE-COUNT.

Answers

1.
```
(DEFINE (ADD* (FLAMBDA (L)
   (ADD-SUB (MAPCAR 'EVAL L))
)))
```

2.
```
(DEFINE (DEFUN (FLAMBDA (L)
   (COND
     ((EQUAL (CADR L) 'FEXPR)
        (EVAL (CONS 'DEFINE
           (LIST (CONS (CAR L)
             (LIST (CONS 'FLAMBDA (CDDR L)))
                 )
        ))
      ))
   (T (EVAL (CONS 'DEFINE
     (LIST (CONS (CAR L)
        (LIST CONS 'LAMBDA (CDR L))
          )
     ))
   ))
   )
)))
```

3.
```
(DEFINE (NEWSETQ (FLAMBDA (L)
   (SETQ (NEWSETQ-USE-COUNT (ADD NEW-SETQ-USE-COUNT 1))
   (EVAL (CONS 'SETQ L))
)))
```

Contents | Control Structures | Properties and Lambda Expressions

LEARNING LISP

Contents | Eval and Apply | Differentiating Polynomials

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, and so on. Each thing in the drawers, however, is still associated with the chest [the atom].

Let's construct our chest of drawers.

```
:(PUT 'CHEST 'TOP '(SOX))

  (SOX )

:(PUT 'CHEST 'SECOND '(UNDERWEAR (SHORT

:SHIRTS))))

  (UNDERWEAR (SHORT 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 that 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 the 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 (SHORT SHIRTS ) )

:(GET 'CHEST 'TOP)

  (SOX )

:(SETQ PLACE 'CHEST)

  CHEST

:(PUT PLACE 'TOP (CONS (GET PLACE 'TOP)

:(GET PLACE 'SECOND)))))

  ((SOX ) UNDERWEAR (SHORT 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 similar to pulling out a drawer. We can't GET the value of that property after it has been REMed.

We said previously 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:

```
:(CDR PLACE)

  (TOP ((SOX ) UNDERWEAR (SHORT SHIRTS ) )
  THIRD (T-SHIRTS JEANS ) BOTTOM (
  PAJAMAS ) )

:(CDR 'CHEST)

  (TOP ((SOX ) UNDERWEAR (SHORT SHIRTS ) )
  THIRD (T-SHIRTS JEANS ) BOTTOM (
  PAJAMAS ) )
```

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

```
:
:(SETQ CHEST 5)

  5

:(CDR 'CHEST)

  (TOP ((SOX ) UNDERWEAR (SHORT SHIRTS ) )
  THIRD (T-SHIRTS JEANS ) BOTTOM (
  PAJAMAS ) )

:CHEST

  5
```

What are properties used for? Why are they in Lisp?

For a simple example we might arrange our phonebook according to our friends' names. Each name has associated with it a property "number" and a property "address". This isn't much 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)

  (ADDRESS (123 FRONT ROADD ) PHONE (
  345 6789 ) )

:(PUT 'DAVE 'ADDRESS '(321 TRONF STREET))

  (321 TRONF STREET )
```

```
:(PUT 'DAVE 'PHONE '(WE7 1212))

  (WE7 1212 )

:(GET 'DAVE 'ADDRESS)

  (321 TRONF STREET )
```

But this is useless because we are restricted to using address parts and phone numbers that are Lisp atoms. Anyway we could have done the whole program 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 'SPEECHPART 'NOUN)

  NOUN

:(PUT 'AARDVARK 'NUMBER 'SINGULAR)

  SINGULAR

:(PUT 'EAT 'SPEECHPART 'VERB)

  VERB

:(PUT 'EAT 'VERBTYPE 'TRANSITIVE)

  TRANSITIVE

:(PUT 'SOUPS 'SPEECHPART 'NOUN)

  NOUN

:(PUT 'SOUPS 'NUMBER 'PLURAL)

  PLURAL
```

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

```
:(DEFINE (PARTS (FLAMBDA (SENTENCE)

:  (MAPCAR '(LAMBDA (WORD) (GET WORD 'SPEECHPART))

:          SENTENCE)))))

  PARTS

:(PARTS EAT AARDVARK SOUPS)

  (VERB NOUN NOUN)
```

What, you may ask, was all that about? It looks like we half-wrote a function in the middle of another one! The expression "(LAMBDA (WORD) . . . 'SPEECHPART))" 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 like 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 S
  PEECHPART ) ) ) ) SENTENCE ) ) )

:(GET 'PARTS 'EXPR)

  (FLAMBDA (SENTENCE ) (MAPCAR (QUOTE (
  LAMBDA (WORD ) (GET WORD (QUOTE
  SPEECHPART ) ) ) ) SENTENCE ) )
```

In general, LISP looks at the world as follows:

1. Everything is an expression [that is, 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 the LAMBDA expression!
5. If the CAR is a list, evaluate it and go to rule 4.

That's all a bit complicated. 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 BE TO 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 whether DEFINE was an EXPR or a FEXPR. Since we know what DEFINE really does, 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 it. 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 to do this, it is a good idea to call it something other than DEFINE [like DEFINA], to avoid making catastrophic mistakes.

```
:
:(DEFINE (DEFINA (FLAMBDA (FUNCFORM)

: (PUT (CAAR FUNCFORM) 'EXPR (CADAR

:    FUNCFORM)))))

  DEFINE

:(DEFINA (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 'DEFINA)

  (EXPR (FLAMBDA (FUNCFORM ) (PUT (CAAR
  FUNCFORM ) (QUOTE EXPR ) (CADAR FUNCFORM
  ) ) ) )

:(REM 'DEFINA 'EXPR)

  NIL

:(CDR 'DEFINA)

  NIL

:(CDR 'DEFINE)

  (SUBR * )
```

Note that when we redefined DEFINE we are using only the value of DEFINA. 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 [IF YOU USE DEFINE INSTEAD OF DEFINA, 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 means that we can write functions which generate other functions during their evaluation.

Contents | Eval and Apply | Differentiating Polynomials

LEARNING LISP

Contents | Properties and Lambda Expressions | Simplifying Polynomials

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.

Here are the rules for differentiation which 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. Notice that some of these rules are recursive. For example, in order to differentiate (u+v) we need to differentiate *u* and *v*. So much for the specification 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 polynomials for use in our derivative function. We will need the outermost [or highest level] function in a 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.

```
:(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. The derivative is 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.

```
:(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. Therefore, 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)) 1))
  (MULT (EXP X 2) 0)) (ADD (MULT B 1) (MULT X 0))
  0) 0)
:(setq a (derv '(mult (add x 1) (sub 1 x)) 'x))
  (ADD (MULT (ADD X 1) (SUB 0 1)) (MULT (SUB 1 X)
  (ADD 1 0)))
:(setq x 3)
  3
:(eval a)
  -6
```

Contents | Properties and Lambda Expressions | Simplifying Polynomials

LEARNING LISP

Contents | Differentiating Polynomials | Efficiency and Elimination of Recursion

Simplifying Polynomials

Let us now return once again to the world of polynomials. We have only one task remaining, 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:

```
:(derv '(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 many 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 behavior.

```
:(simplify '(add x 0))
  X
:(simplify '(exp 0 0))
```

```
  1
:(simplify '(mult (mult 0 x) y))
  0
:(setq p (derv '(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.

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 closely follows the list of semplifications given 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 semplification. 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.

This is one of those cases, when recursion isn't the easiest way to do things. Thus, let's use a PROG! The following 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.

Contents | Differentiating Polynomials | Efficiency and Elimination of Recursion

LEARNING LISP

Contents | Simplifying Polynomials | ELIZA

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 prod m))
:    (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)
:  )
:)))

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 setp 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 [that is, 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.

So much for factorial. We mentioned several times that Lisp was an *interpreter*. What does this mean?

Language processors come in two different types, 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 waste of time. This 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 tipically 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.

Contents | Simplifying Polynomials | ELIZA

LEARNING LISP

Contents | Efficiency and Elimination of Recursion | The P-Lisp Interpreter

ELIZA

ELIZA is a classic example of the use of Lisp in a sample artificial intelligence (AI) application. Originally written by Joseph Weizenbaum, ELIZA is a program that "simulates" a patient's discussion with a psychiatrist. While appearing on the surface to possess "intelligence", the program is actually little more than a simple pattern matcher. A sample dialogue with ELIZA follows (user input is in lower case, with ELIZA's responses in upper case):

```
  MAY I HELP YOU?
:i want you to help me
  WHAT WOULD IT MEAN TO YOU IF YOU GOT ME TO HELP YOU?
:im afraid to tell you
  YOURE NOT REALLY TALKING ABOUT ME, ARE YOU?
:what does that mean
  WHY DO YOU ASK?
:are you asking me
  WHY ARE YOU INTERESTED IN WHETHER OR NOT I AM ASKING YOU?
:why do you keep answering with questions
  DOES THAT QUESTION INTEREST YOU?
:i dont know
  DONT YOU REALLY KNOW?
:no
  ARE YOU SAYING NO JUST TO BE NEGATIVE?
:i think so
  DO YOU REALLY THINK SO?
:not always
  CAN YOU THINK OF A SPECIFIC EXAMPLE?
:oh shut up
  IM NOT SURE I UNDERSTAND YOU FULLY?
:goodbye
  GOODBYE
```

What follows is an annotated listing of ELIZA as implemented in p-Lisp:

```
;
; ELIZA -- P-Lisp version Copyright 1982 by Steven Cherry
;
; -----
; The main function, ELIZA, gets a line of input and prints the
; appropriate response. The user's input is saved for the next time
; through the loop so we can check if the user is repeating himself.
;
```

```
(DEFINE (ELIZA (LAMBDA ()
  (PROG (KEY IN ON)
    (CALL -936)                          ;clear the screen
    (GC NIL)                             ;turn off garbage collection message
    (PRINT)
    (PRINTLINE '(MAY I HELP YOU?))
    LOOP
      ; Get input. If the new input equals the old, complain.
      ; Otherwise check if input should stop ELIZA.
      (SETQ IN (READLINE))
      (COND
        ((EQUAL IN ON)
            (PRIN1)
            (PRINTLINE '(PLEASE DO NOT REPEAT YOURSELF))
            (GO LOOP))
        ((EQUAL IN '(GOODBYE)) (RETURN 'GOODBYE)))
      (SETQ KEY (KEYSEARCH IN))          ;get the keyword number
      (PRIN1)
      (PRINTLINE (REPLY (CONJUGATE (CDR KEY))
                        (CAR KEY)))      ;print response
      (SETQ IN ON)
    (GO LOOP)
  )
)))

;
; KEYSEARCH searches the input line for the word or phrase
; that has the highest priority. This priority is returned, along
; with the remainder of the input line.
;
; Priorities are stored on a word's property list under the property
; KEY. If the property value is not a number, then the word can be part
; of a phrase, in which case the property value will also appear as a
; property on the list. The value of this property will be either the
; priority for the phrase or the next word of the phrase. If a word can
; have a priority by itself as well as be the first word of a phrase
; then this priority will be stored under the property KEY2. As an
; example, the property list for the word YOU might look like:
;     (KEY ARE ARE 4 KEY2 14),
; meaning the word YOU has priority 14, and the phrase YOU ARE has
; priority 4.
;
(DEFINE (KEYSEARCH (LAMBDA (IN)
  (PROG (KEYNUM THISKEY LEFT WORD)
    (SETQ KEYNUM 1 THISKEY 1)
    LOOP
      ; If at end of input, return the highest priority and the input
      ; following the keyword or keyphrase selected.
```

```
      (COND ((NULL IN) (RETURN (CONS KEYNUM LEFT))))
      (SETQ WORD (CAR IN))
      ; Check if the word is a number, since doing a GET on a numeric
      ; atom causes an error. Set THISKEY to the property value of
      ; KEY (could be NIL if the word isn't a keyword).
      (COND
        ((NUMBER WORD) (SETQ THISKEY 1))
        (T (SETQ THISKEY (GET WORD 'KEY))))
      ; If THISKEY is not numeric, the word must be the first word
      ; of a keyphrase.
      (COND
        ((NOT (NUMBER THISKEY))
          (SETQ THISKEY (KEYPHRASE WORD (CDR IN) THISKEY))
          ; If KEYPHRASE returned NIL, the phrase wasn't in the input.
          ; In this case get the KEY2 property value. Otherwise,
          ; KEYPHRASE returned the priority of the phrase and the
          ; input following the phrase.
          (COND
            ((NULL THISKEY)
                (SETQ IN (CDR IN) THISKEY (GET2 WORD 'KEY2)))
            (T (SETQ IN (CDR THISKEY) THISKEY (CAR THISKEY)))))
        (T (SETQ IN (CDR IN))))
      ; If the new priority is higher than the old one, set the old
      ; to the new and set LEFT to contain the remainder of the input.
      (COND
        ((GREATER THISKEY KEYNUM)
          (SETQ KEYNUM THISKEY LEFT IN)))
    '(GO LOOP)
  )
)))

;
; KEYPHRASE checks for a specific sequence of words in IN. If found,
; the priority of the phrase and the input following the phrase are
; returned as the CAR and CDR of a list respectively.
;
(DEFINE (KEYPHRASE LAMBDA (WORD IN THISKEY)
  (PROG ()
    LOOP
      (COND
        ((NULL THISKEY) (RETURN NIL))        ;check if finished
        ; Return the priority for the phrase (in THISKEY)
        ; has been found.
        ((NUMBER THISKEY) (RETURN (CONS THISKEY IN)))
        ; Use the next word in the input as a property on WORD's
        ; property list, get the property value and loop.
        (T (SETQ THISKEY (GET WORD (CAR IN)) IN (CDR IN))))
    (GO LOOP)
```

```
  )
)))

;
; GET2 returns the property value of property Y for atom X. If this
; value is NIL, then GET2 returns 1.
;
(DEFINE (GET2 (LAMBDA (X Y)
  (PROG (Z)
    (SETQ Z (GET X Y))
    (COND
      (NULL Z) (RETURN 1))
      (T (RETURN Z)))
  )
)))

;
; CONJUGATE conjugates the input line following the keyword or phrase
; found by KEYSEARCH. The conjugation of a word is stored on the word's
; property list under the property CONJ. For example, I is changed to
; YOU, YOU is changed to ME, etc. The resulting list is returned.
;
(DEFINE (CONJUGATE (LAMBDA (OLDT)
  (PROG (NEW W W2)
    LOOP
      (COND
        ((NULL OLDT) (RETURN NEW)))        ; return if finished
      (SETQ W (CAR OLDT))
      ; If W is a word, get the value of the CONJ property
      (COND
        ((NUMBER W) (SETQ W2 NIL))
        (T (SETQ W2 (GET W 'CONJ))))
      ; If W2 is non-NIL, it will be the conjugation of word W. Append
      ; the correct word onto NEW and repeat.
      (COND
        ((NULL W2) (SETQ NEW (APPEND NEW W)))
        (T (SETQ NEW (APPEND NEW W2))))
      (SETQ OLDT (CDR OLDT))
    (GO LOOP)
  )
)))

;
; REPLY is given the priority of the found keyword or phrase (KEYNUM) and
; the conjugated remainder of the input line (NEW) and formulates a
; response. First the priority is used to select the response set from
; the RESP list. The CAR of this response set is used to select the next
; response in the set, and this number is then incremented. If the
```

```
; number exceeds the length of the set, it is reset to 2. Once a
; response is selected, NEW is attached to the end if the CAR of the
; response is an asterisk.
;
(DEFINE (REPLY (LAMBDA (NEW KEYNUM)
  (PROG (A RES OUT)
    ; Use KEYNUM to select the response set
    (SETQ A (ARRAY (LIST KEYNUM) RESP))
    (SETQ RES (EVAL A))
    ; Use the CAR of the response set to select the
    ; next response.
    (SETQ OUT (ARRAY (LIST CAR RES)) (CDR RES)))
    ; Reset the number to 2 if nothing was selected, and
    ; select the first response. Otherwise increment the number.
    (COND
      ((NULL OUT)
        (SET A (CONS 2 (CDR RES)))
        (SETQ OUT (CAR (CDR RES))))
      (T (SET A (CONS (ADD (CAR RES) 1) (CDR RES)))))
    ; If the CAR of the response is an asterisk, add NEW to the end.
    (COND
      ((EQUAL (CAR OUT) '*)
        (SETQ OUT (CONC (CDR OUT) NEW))))
    (RETURN OUT)
  )
)))

;
; PRINTLINE prints a list without the delimiting parentheses.
;
(DEFINE (PRINTLINE (LAMBDA (X)
  (MAPCAR 'PRIN1 X)
  (PRIN1)
)))

;
;
;
; ELIZA DATABASE
;
(SETQ RESP '(R1 R2 R3 R4 R5 R6 R7 R8 R9 R10 R11 R12 R13 R14 R15
  R16 R16 R16 R16 R16 R16 R22 R23 R24 R25 R26 R27 R28 R29 R30
  R31 R32 R33 R34 R35 R36)))

(SETQ R1 '(1
  (WHAT DOES THAT SUGGEST TO YOU?)
  (I SEE)
  (IM NOT SURE I UNDERSTAND YOU FULLY)
```

```
  (COME, COME, ELUCIDATE YOUR THOUGHTS)
  (CAN YOU ELABORATE ON THAT?)
  (THAT IS QUITE INTERESTING)
  (DO YOU HAVE ANY PSYCHOLOGICAL PROBLEMS?)
  (YOU DONT SAY)
))

(SETQ R2 '(1
  (* DONT YOU BELIEVE THAT I CAN)
  (* PERHAPS YOU WOULD LIKE TO BE ABLE TO)
  (* YOU WANT ME TO BE ABLE TO)
  (* WHAT MAKES YOU THINK I COULD)
))

(SETQ R3 '(
  (* PERHAPS YOU DONT WANT TO)
  (* DO YOU WANT TO BE ABLE TO)
  (* DO YOU THINK YOU COULD NOT)
))

(SETQ R4 '(1
  (* WHAT MAKES YOU THINK I AM)
  (* DOES IT PLEASE YOU TO BELIEVE I AM)
  (* PERHAPS YOU WOULD LIKE TO BE)
  (* DO YOU SOMETIMES WISH YOU WERE)
))

(SETQ R5 '(1
  (* WHAT MAKES YOU THINK I AM)
  (* DOES IT PLEASE YOU TO BELIEVE I AM)
  (* PERHAPS YOU WOULD LIKE TO BE)
  (* DO YOU SOMETIMES WISH YOU WERE)
))

(SETQ R6 '(1
  (* DONT YOU REALLY)
  (* WHY DONT YOU)
  (* DO YOU WISH TO BE ABLE TO)
  (* DOES THAT TROUBLE YOU?)
))

(SETQ R7 '(1
  (TELL ME MORE ABOUT SUCH FEELINGS)
  (* DO YOU OFTEN FEEL)
  (* DO YOU ENJOY FEELING)
))

(SETQ R8 '(1
```

```
  (* DO YOU REALLY BELIEVE I DONT)
  (* PERHAPS IN GOOD TIME I WILL)
  (* DO YOU WANT ME TO)
))

(SETQ R9 '(1
  (* DO YOU THINK YOU SHOULD BE ABLE TO)
  (* WHY CANT YOU)
))

(SETQ R10 '(1
  (* WHY ARE YOU INTERESTED IN WHETHER OR NOT I AM)
  (* WOULD YOU PREFER IF I WERE NOT)
  (* PERHAPS IN YOUR FANTASIES I AM)
))

(SETQ R11 '(1
  (* HOW DO YOU KNOW YOU CANT)
  (HAVE YOU TRIED?)
  (* PERHAPS YOU CAN NOW)
))

(SETQ R12 '(1
  (* DID YOU COME TO ME BECAUSE YOU ARE)
  (* HOW LONG HAVE YOU BEEN)
  (* DO YOU BELIEVE IT IS NORMAL TO BE)
  (* DO YOU ENJOY BEING)
))

(SETQ R13 '(1
  (* DID YOU COME TO ME BECAUSE YOU ARE)
  (* HOW LONG HAVE YOU BEEN)
  (* DO YOU BELIEVE IT IS NORMAL TO BE)
  (* DO YOU ENJOY BEING)
))

(SETQ R14 '(1
  (WE WERE DISCUSSING YOU -- NOT ME)
  (* OH, I)
  (YOURE NOT REALLY TALKING ABOUT ME, ARE YOU?)
  (OH, YEAH?)
))

(SETQ R15 '(1
  (* WHAT WOULD IT MEAN TO YOU IF YOU GOT)
  (* WHY DO YOU WANT)
  (* SUPPOSE YOU SOON GOT)
  (* WHAT IF YOU NEVER GOT)
```

```
  (* I SOMETIMES ALSO WANT)
))

(SETQ R16 '(1
  (WHY DO YOU ASK?)
  (DOES THAT QUESTION INTEREST YOU?)
  (WHAT ANSWER WOULD PLEASE YOU MOST?)
  (WHAT DO YOU THINK?)
  (ARE SUCH QUESTIONS ON YOUR MIND OFTEN?)
  (WHAT IS IT THAT YOU REALLY WANT TO KNOW?)
  (HAVE YOU ASKED ANYONE ELSE?)
  (HAVE YOU ASKED SUCH QUESTIONS BEFORE?)
  (WHAT ELSE COMES TO MIND WHEN YOU ASK THAT?)
  (ARE YOU ASKING ME?)
))

(SETQ R22 '(1
  (NAMES DONT INTEREST ME)
  (I DONT CARE ABOUT NAMES -- PLEASE GO ON)
))

(SETQ R23 '(1
  (IS THAT THE REAL REASON?)
  (DONT ANY OTHER REASONS COME TO MIND?)
  (DOES THAT REASON EXPLAIN ANYTHING ELSE?)
  (WHAT OTHER REASONS MIGHT THERE BE?)
))

(SETQ R24 '(1
  (PLEASE DONT APOLOGIZE)
  (APOLOGIES ARE NOT NECESSARY)
  (WHAT FEELINGS DO YOU HAVE WHEN YOU APOLOGIZE)
  (DONT BE SO DEFENSIVE)
))

(SETQ R25 '(1
  (WHAT DOES THAT DREAM SUGGEST TO YOU?)
  (DO YOU DREAM OFTEN?)
  (WHAT PERSONS APPEAR IN YOUR DREAMS?)
  (ARE YOU DISTURBED BY YOUR DREAMS?)
))

(SETQ R26 '(1
  (HOW DO YOU DO -- PLEASE STATE YOUR PROBLEM)
  (ENOUGH SALUTATIONS -- WHAT DO YOU WANT?)
))

(SETQ R27 '(1
```

```
  (HOW DO YOU DO -- PLEASE STATE YOUR PROBLEM)
  (ENOUGH SALUTATIONS -- WHAT DO YOU WANT?)
))

(SETQ R28 '(1
  (YOU DONT SEEM QUITE CERTAIN)
  (WHY THE UNCERTAIN TONE?)
  (CANT YOU BE MORE POSITIVE?)
  (YOU ARENT SURE?)
  (DONT YOU KNOW?)
))

(SETQ R29 '(1
  (ARE YOU SAYING NO JUST TO BE NEGATIVE?)
  (YOU ARE BEING A BIT NEGATIVE)
  (WHY NOT?)
  (ARE YOU SURE?)
  (WHY NO?)
))

(SETQ R30 '(1
  (* WHY ARE YOU CONCERNED ABOUT MY)
  (* WHAT ABOUT YOUR OWN)
))

(SETQ R31 '(1
  (CAN YOU THINK OF A SPECIFIC EXAMPLE?)
  (WHEN?)
  (WHAT ARE YOU THINKING OF?)
  (REALLY, ALWAYS?)
))

(SETQ R32 '(1
  (DO YOU REALLY THINK SO?)
  (* BUT YOU ARE NOT SURE YOU)
  (* DO YOU DOUBT YOU)
))

(SETQ R33 '(1
  (IN WHAT WAY?)
  (WHAT RESEMBLANCE DO YOU SEE?)
  (WHAT OTHER CONNECTIONS DO YOU SEE?)
  (HOW?)
))

(SETQ R34 '(1
  (YOU SEEM QUITE POSITIVE)
  (ARE YOU SURE?)
```

```
  (I SEE)
  (I UNDERSTAND)
))

(SETQ R35 '(1
  (WHY DO YOU BRING UP THE TOPIC OF FRIENDS?)
  (DO YOUR FRIENDS WORRY YOU?)
  (ARE YOU SURE YOU HAVE ANY FRIENDS?)
  (DO YOUR FRIENDS PICK ON YOU?)
))

(SETQ R36 '(1
  (DO COMPUTERS WORRY YOU?)
  (ARE YOU TALKING ABOUT ME IN PARTICULAR?)
  (ARE YOU FRIGHTENED BY MACHINES?)
  (WHY DO YOU MENTION COMPUTERS?)
  (WHAT DO YOU THINK MACHINES HAVE TO DO WITH YOUR PROBLEM?)
  (DONT YOU THINK COMPUTERS CAN HELP PEOPLE?)
  (WHAT IS IT ABOUT MACHINES THAT WORRIES YOU?)
))

;
; ELIZA DICTIONARY
;
(PUT 'I 'CONJ 'YOU)
(PUT 'I 'KEY 'DONT)
(PUT 'I 'DONT 6)
(PUT 'I 'FEEL 7)
(PUT 'I 'CANT 11)
(PUT 'I 'AM 12)
(PUT 'I 'WANT 15)
(PUT 'YOURSELF 'CONJ 'MYSELF)
(PUT 'ARE 'CONJ 'AM)
(PUT 'ARE 'KEY 'YOU)
(PUT 'ARE 'YOU 10)
(PUT 'AM 'CONJ 'ARE)
(PUT 'WERE 'CONJ 'WAS)
(PUT 'WAS 'CONJ 'WERE)
(PUT 'YOU 'CONJ 'ME)
(PUT 'YOU 'KEY 'ARE)
(PUT 'YOU 'ARE 4)
(PUT 'YOU 'KEY2 14)
(PUT 'YOUR 'CONJ 'MY)
(PUT 'YOUR 'KEY 30)
(PUT 'MY 'CONJ 'YOUR)
(PUT 'IVE 'CONJ 'YOUVE)
(PUT 'YOUVE 'CONJ 'IVE)
(PUT 'IM 'CONJ 'YOURE)
```

```
(PUT 'IM 'KEY 13)
(PUT 'YOURE 'CONJ 'IM)
(PUT 'YOURE 'KEY 5)
(PUT 'ME 'CONJ 'YOU)
(PUT 'CAN 'KEY 'YOU)
(PUT 'CAN 'YOU 2)
(PUT 'CAN 'I 3)
(PUT 'WHY 'KEY 'DONT)
(PUT 'WHY 'DONT 'YOU)
(PUT 'WHY 'YOU 8)
(PUT 'WHY 'CANT 'I)
(PUT 'WHY 'I 9)
(PUT 'WHY 'KEY2 21)
(PUT 'WHAT 'KEY 16)
(PUT 'HOW 'KEY 17)
(PUT 'WHO 'KEY 18)
(PUT 'WHERE 'KEY 19)
(PUT 'WHEN 'KEY 20)
(PUT 'NAME 'KEY 22)
(PUT 'NAMES 'KEY 22)
(PUT 'CAUSE 'KEY 23)
(PUT 'BECAUSE 'KEY 23)
(PUT 'SORRY 'KEY 24)
(PUT 'DREAM 'KEY 25)
(PUT 'DREAMS 'KEY 25)
(PUT 'HELLO 'KEY 26)
(PUT 'HI 'KEY 27)
(PUT 'MAYBE 'KEY 28)
(PUT 'NO 'KEY 29)
(PUT 'ALWAYS 'KEY 31)
(PUT 'THINK 'KEY 32)
(PUT 'ALIKE 'KEY 33)
(PUT 'YES 'KEY 34)
(PUT 'FRIEND 'KEY 35)
(PUT 'FRIENDS 'KEY 35)
(PUT 'COMPUTER 'KEY 36)
(PUT 'MACHINE 'KEY 36)
(PUT 'MACHINES 'KEY 36)
(PUT 'COMPUTERS 'KEY 36)
```

Contents | Efficiency and Elimination of Recursion | The P-Lisp Interpreter

LEARNING LISP

Contents | ELIZA | Appendix: The Lisp Editor

The P-Lisp Interpreter

This chapter is intended for those who are curious about how a Lisp interpreter manages to do all the wonderful things described in the previous chapters. The chapter is a bit technical in nature and assumes the reader has a basic knowledge of bits, bytes, and similar aspects of computer internals. Although written specifically about the P-Lisp system, many of the concepts and methods employed here are used in many different Lisp systems.

On its most fundamental level, a Lisp interpreter consists of little more than a set of routines to handle some fancy pointer manipulation (a pointer being an address of some location in memory). This simplicity is a direct result of the uniformity of Lisp data structures. As described below, the primary data structure, the list, maps quite easily onto an equivalent internal representation. The interpreter's simplicity is further enhanced by the non-necessity of a sophisticated parser, due to Lisp's rather simple syntax. Moreover, because Lisp is by definition a recursive language, the interpreter may be defined recursively as well, substantially reducing its complexity.

The P-Lisp workspace is divided into four-byte units called "cells." The first two bytes of a cell are called (naturally enough) the CAR of the cell, and the last two bytes are the cell's CDR. The cells are aligned on contiguous four-byte boundaries, so that the last two bits of a cell's address are always zero (see below). This is necessary because the last two bits may then be used as status flags or to describe a cell's contents.

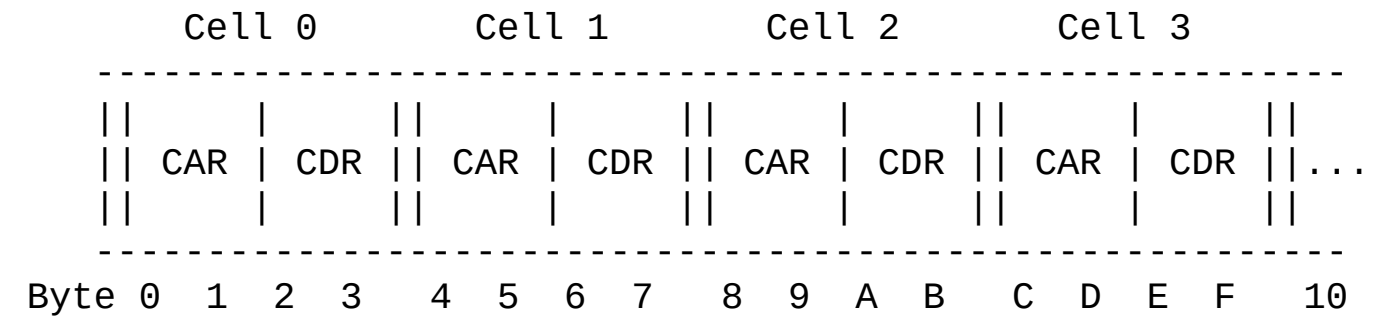


Figure 27.1. Memory Cells

For example, the CAR or CDR of a cell will typically contain the address of another cell. Bit 1 of such a pointer is used to indicate the type of data it is pointing at. If the bit is set, the pointer (with the bit reset) points to an atom; if the bit is reset, the pointer points to a list. A list is represented as a linked list of

cells. For example, the list (A B C) is represented simply as

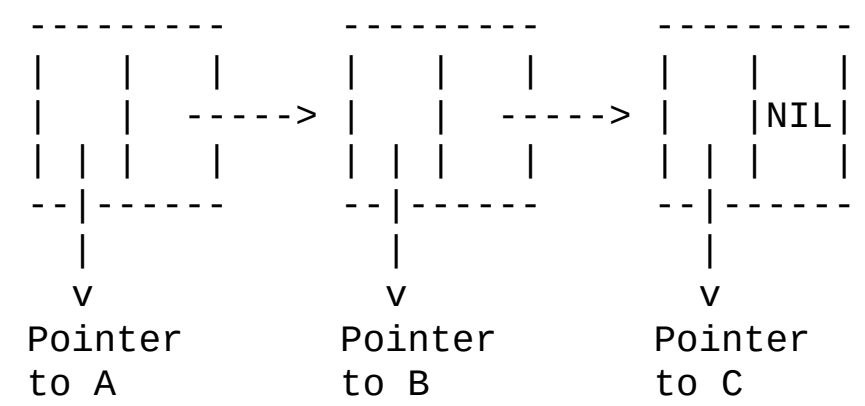


Figure 27.2. List Representation

The CDR of the last box is a pointer to the atom NIL. To make life easy for the interpreter, NIL is predefined to live at a location in memory where the hi-byte of its address is guaranteed to be zero (such an address is said to reside in "page zero" of memory). So, if the first byte of a pointer is zero, the interpreter assumes that the pointer points to NIL, regardless of the value of the second byte.

Note in Figure 27.2 how the list (A B C) maps directly onto its internal representation. The CAR of (A B C) is A, and the CAR of the first cell of the list points to the atom A. The CDR of (A B C) is (B C), and the CDR of the first cell points to the internal representation of (B C). You should be able to see already that to evaluate (CAR '(A B C)), for example, the CAR subroutine simply returns the CAR of the first cell of (A B C).

P-Lisp supports three types of atoms: literal atoms, integer atoms, and floating-point atoms. Literal atoms are stored with the following format:

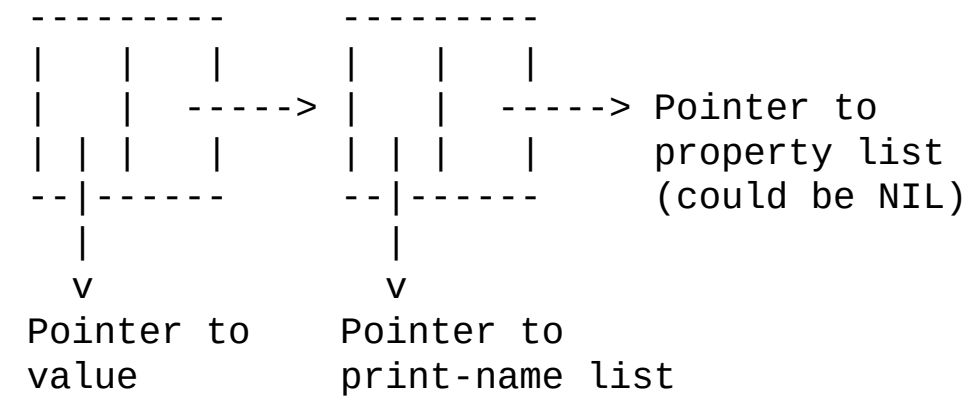


Figure 27.3. Literal Atom

A literal atom's print name (the sequence of characters comprising the name of the atom) is stored in a linked list. The CAR of each cell in this list contains two characters of the atom's name, while the CDR

points to the next cell in the list. The CAR of an atom's first cell points to the atom's value; if the atom has no value, this pointer is zero. Note that in the latter case both bytes are zero; this is necessary to distinguish the no-value case from the NIL-value case, which has only the hi-byte zero.

As an example, the atom APPLE with the value FRUIT and the property list (COLOR RED) would have the following format:

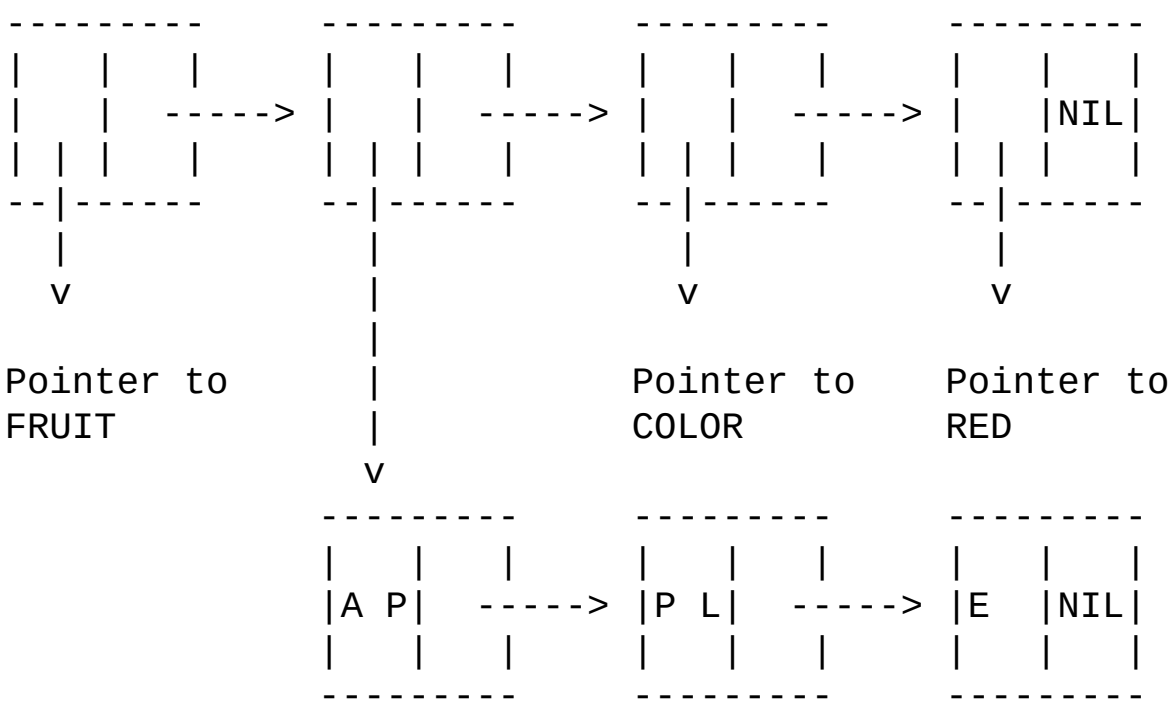


FIgure 27.4. Literal Atom

Numeric atoms are distinguished from literal atoms by having the CDR of the first cell NIL, with the CAR pointing to the remainder of the atom. An integer atom has the following format:

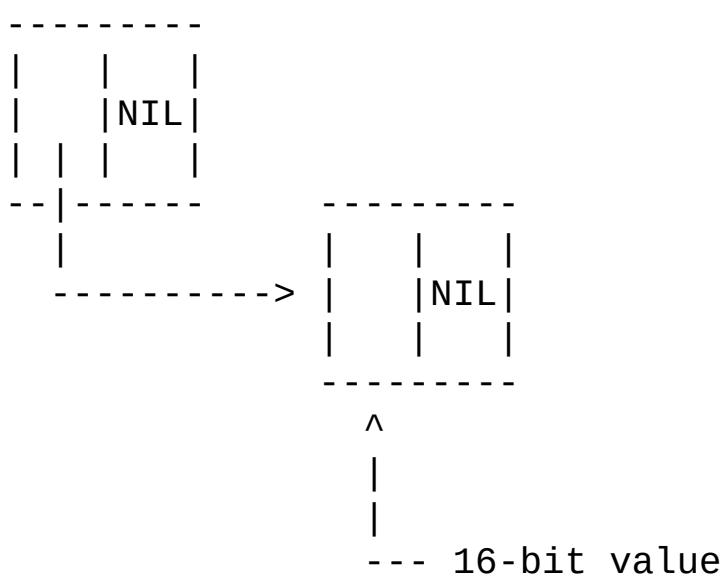


Figure 27.5. Integer Atom

The 16-bit value is in two's-complement form. The NIL in the CDR of the second cell distinguishes integer atoms from floating-point atoms. A floating-point atom has the following format:

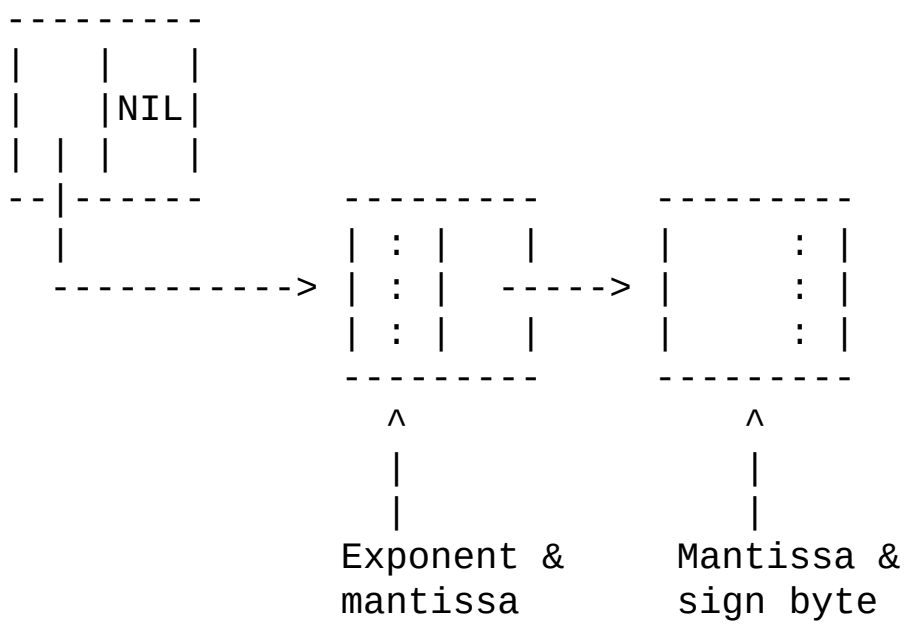


Figure 27.6. Floating-point Atom

Floating-point atoms are stored in unpacked normalized exponential form. The mantissa is four bytes in length, providing ten significant digits.

All literal atoms, including the built-in atoms (T, NIL, and the built-in functions), are organized into a single list within the workspace, called the OBLIST. Literal atoms are always unique; for some atom A, there can be only one instance of A in the workspace. All references to this atom are pointers to the single location where A resides. Numeric atoms, however, are not unique; there can be any number of instances of a given numeric value.

When the Lisp READ routine reads a s-expr, the s-expr is parsed and an internal representation of the s-expr is built. For every literal atom, the interpreter scans the OBLIST for that atom, comparing the atom's print name to those of the atom on the OBLIST. If the atom is found, the pointer to the atom (with the atom bit set) is returned; otherwise, an internal representation of the new atom is built and added to the end of the OBLIST, and a pointer to the new atom is returned. If READ reads a numeric atom, it simply builds a new atom (integerized, if possible) and returns its pointer. Although unique numeric atoms would provide a substantial memory savings, vital for any microcomputer implementation, the result would be a severe degradation of performance in any application involving extensive calculations (consider that, if numeric atoms were unique, the OBLIST would have to be scanned after every calculation to determine if the result already lives on the OBLIST).

After READ builds the linked list of cells representing the just-read s-expr, a pointer to the list is passed

on to the evaluator, EVAL. EVAL determines if the s-expr is an atom or a list by the status of the atom bit. If the bit is set, the s-expr is an atom, and EVAL evaluates it in the following way: If the atom is numeric (indicated by the CDR of the first cell being NIL), EVAL just returns the pointer to the atom as the result. In the case of a literal atom, EVAL first scans the environment chain (see discussion on environment chain below) for the atom, and returns the corresponding value if found. If the atom is not on the chain, EVAL uses the atom's OBLIST entry, returning the CAR of the first cell as its value.

If EVAL is handed a list to evaluate, the CAR of the first cell is used to determine the function to be applied. If the pointer points to an atom, EVAL scans the property list of the atom for the EXPR property. If found, the corresponding property value, the function definition, is evaluated, with the CDR of the first cell handed to EVAL used as the pointer to the argument list. If EVAL does not find the EXPR property, the property list is scanned again, this time searching for the SUBR property. If this property is found, the corresponding property value is used as the address of the interpreter subroutine that evaluates this SUBR. This routine is called, passing to it the pointer to the argument list. If the CAR of the first cell handed to EVAL points to a list, EVAL calls itself to evaluate this list, applying the end result of this evaluation (which must be an EXPR or SUBR) to the argument list.

The SUBR routines themselves consist primarily of building new cells, deleting old cells, and moving pointers around. Consider, for example, the function CONS. Recall that (CONS 'A '(B)) creates a new list (A B), the CAR of which is A, and the CDR of which is (B). This is accomplished in the interpreter by simply getting a new cell and storing the pointer to A in the CAR and the pointer to (B) in the CDR. The pointer to the new cell now points to the result, the list (A B).

The environment chain contains the active LAMBDA-bindings during a given evaluation. LAMBDA-bindings are stored as cell pairs; the CAR of the first cell points to the formal argument, and the CAR of the second cell points to the value bound to the formal argument. The end of the environment is marked by a NIL formal argument, with the CDR of that cell pointing to the next environment on the chain. Below is a sample environment chain.

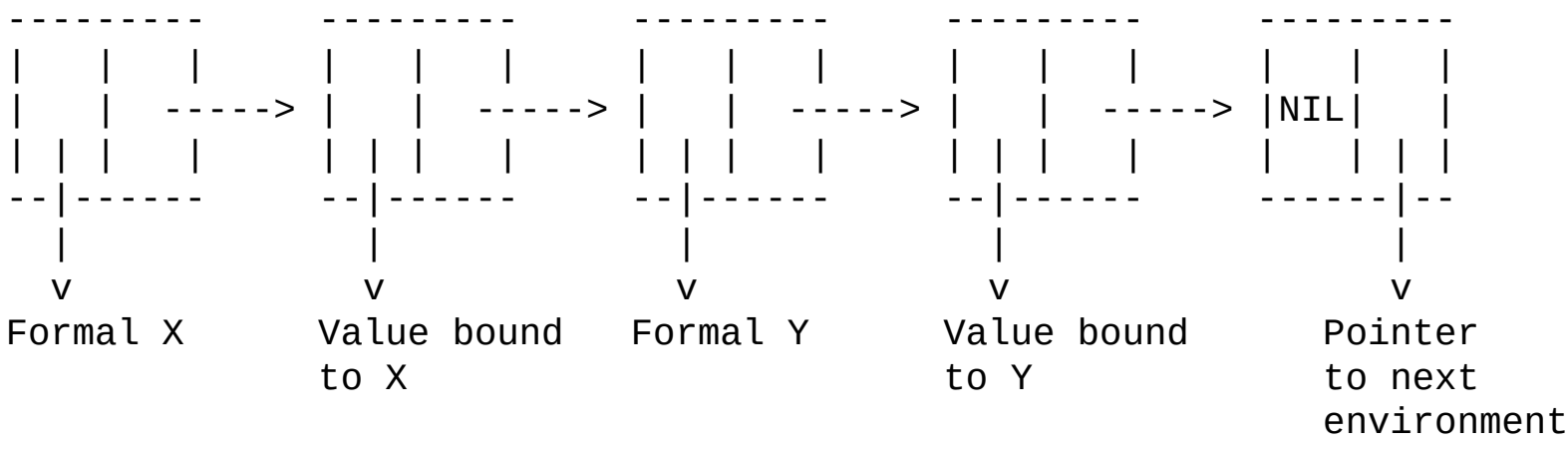


Figure 27.7. Environment Chain

Whenever an environment is exited (for example, leaving a PROG or a LAMBDA-expression), the cells

in the environment are discarded and the environment chain pointer is set to point to the next environment on the chain. Whenever a new environment is entered (a new invocation of a LAMBDA-expression or PROG), a new environment is built and attached to the head of the environment chain.

During the course of an evaluation, cells are constantly being used and discarded as required by the interpreter, for example, by building and discarding environments. Initially, all free cells in a workspace are organized into a linked list called the free-space list. New cells are taken from this list as needed by the interpreter. When the interpreter runs out of new cells, a routine called the Garbage Collector is invoked. The Garbage Collector scans the entire workspace, collecting all discarded cells into a new free-space list. This collection is accomplished in two phases, the Mark phase and the Sweep phase. During the Mark phase, all cells that are currently "active," that is, those cells that are attached to the OBLIST or those that are part of the environment or the recursion stack, are marked as active. This is done by setting the garbage collector bit, bit 0, of the CDR of each cell. Cells that are not marked at the end of the Mark phase cannot be reached via a pointer path either through the OBLIST, the environment chain, or the recursion stack; these cells are considered free and reusable. During the Sweep phase of garbage collection, the free cells are collected into a new free-spacelist. This is accomplished by scanning the workspace from top to bottom. All cells that are marked are simply unmarked, while cells that are initially unmarked are added to the freespace list. After this phase is completed, the interpreter continues processing where it left off when the Garbage Collector was invoked.

SUMMARY

In very broad terms we have described the basic workings of the P-Lisp interpreter. A more detailed examination would probably be beyond the scope of this book. However, you should now be able to reread the tutorial with a better understanding of how and why things work the way they do. If you wish to know more about the design of Lisp interpreters in general, John Allen's "Anatomy of Lisp" is an excellent source for such information.

Contents | ELIZA | Appendix: The Lisp Editor

LEARNING LISP

Contents | The P-Lisp Interpreter

Appendix: The Lisp Editor

This appendix includes a detailed explanation of the Lisp editor [ED] including all the code with comments. It is certainly worth going through with a fine-tooth-comb and figuring out how ED does what it does. It really embodies all of the things that are discussed in this text. It would make a very nice exercise to add a few commands to the editor yourself. We have not included the pretty-printer code. That is really a separate set of functions. The editor simply calls the pretty printer directly.

```
;
; The lisp editor. Written for P-Lisp by Jeff Shrager
;
; -----
; The main function is ED. It is a FEXPR so that the user can type
; (ED name) without having to quote the name. All that ED itself does
; is to check that the named function really exists [has an EXPR] and
; then call ED-SUB on the function's body. If there is no EXPR then ED
; yells and quits.
;
(DEFINE (ED (FLAMBDA (N) ; N will be a list of the function name
 (PROG (BODY)
  (GC NIL) ; Turn off garbage collection messages
  (SETQ N (CAR N)) ; Fix the arg from being a list to just the name
; If there's nothing in the EXPR property of the named symbol then
; yell.
  (COND ((NULL (SETQ BODY (GET N 'EXPR)))
         (RETURN ' "NO FUNCTION DEFINITION."))
; If a NIL comes back from ED-SUB then the edit was aborted so tell the
; user to confirm. Otherwise, the result is bound to BODY and that gets
; replaced in the EXPR property of the named symbol by the T expression.
        ((NULL (SETQ BODY (ED-SUB BODY)))
         (RETURN ' "EDIT ABORTED."))
        (T (PUT N 'EXPR BODY))
  )
 )
)))
;
; ED-SUB does all the real work. It takes any expression and returns it
```

```
; edited as per command. If an ABORT command is given, ED-SUB returns
; NIL thus indicating to ED that an ABORT was performed.
;
(DEFINE (ED-SUB (LAMBDA (BODY)
 ; POV starts out NIL by virtue of the way PROG works.
 ; WINDOW is simply used to speed up processing so that that display
 ; does not have to be recalculated all the time. COMMAND holds the
 ; command for processing.
 (PROG (WORK WINDOW POV COMMAND)
 ; Loop to EDPRINT in order to redisplay the window. It is recomputed
 ; here also.
  EDPRINT
          ; Print the little "where am I" display.
          (ED-POV-PRINT POV)
          ; Print the window in compressed form and put it in WINDOW.
          ; Although it gets printed compressed, the function returns
          ; the full form for the SETQ.
          (SETQ WINDOW (ED-PRINT BODY POV))
 ; Loop here to reinput a command if errors occur that do not require
 ; recomputing or redisplaying the window.
   EDREAD
       ; Read a command word and jam it in COMMAND. Note that the
        ; individual functions have to (read) their own arguments.
        (SETQ COMMAND (READ))
        ; This is the main command decision structure.
               ; If the command was numeric then movement is attempted.
        (COND ((NUMBER COMMAND)
                    ; If the number is 0 then this is an UP command.
                    ; Make sure that there is someplace to go and then
                    ; simply lop the end off the POV.
              (COND ((ZERO COMMAND)
                    (COND ((NULL POV) (PRINT ' "NO UP FROM HERE.")
                                      (GO EDREAD))
                          (T (SETQ POV (ED-DETAIL POV))
                             (GO EDPRINT))
              ))
             ; Not a 0 so fix negatives (ED-FIXNUM) to the
             ; corresponding positive and the add that to the
             ; POV. ED-FIXNUM will return () if the number is
             ; not a legal element of the window.
             ((NULL (SETQ COMMAND (ED-FIXNUM WINDOW COMMAND)))
              (GO EDREAD))
             ; If no errors occurred then we'll get here with a
             ; sure positive and legal position number. One
```

```
        ; last check -- not an atom! (Can't go to those.)
        (T (COND ((ATOM (ED-NTH WINDOW COMMAND))
                  (PRINT ' "CANNOT GO THERE.")
                  (GO EDPRINT))
           )
           ; Okay -- add the number to the POV and return.
           (SETQ POV (CONC POV (LIST COMMAND)))
           (GO EDPRINT))
 ))
; Print command. An easy one.
((EQUAL COMMAND 'P)
 (PP WINDOW) (PRINT) (GO EDPRINT))
; Delete command. Gets one arg, fixes it, then replaces
; the POV with the window less the deleted element.
((EQUAL COMMAND 'D)
 (COND ((SETQ WORK (ED-FIXNUM WINDOW (READ)))
        (SETQ BODY (ED-REPLACE BODY POV
                               (ED-DELETE WINDOW WORK)))))
 (GO EDPRINT))
; Next command
((EQUAL COMMAND 'NX)
 (SETQ POV (ED-NEXT POV BODY)) (GO EDPRINT))
; Back command.
((EQUAL COMMAND 'BK)
 (SETQ POV (ED-BACK POV BODY)) (GO EDPRINT))
; Insert is much like delete but it reads WHAT, HOW and
; WHERE and does some checking first. All that's done in
; ED-INSERT-CHECK which returns () if somethings wrong.
((EQUAL COMMAND 'I)
 (SETQ WORK (ED-INSERT-CHECK WINDOW (READ) (READ) (READ)))
 (COND ((NULL WORK) (GO EDREAD))
        (T (SETQ BODY (ED-REPLACE BODY POV WORK))
           (GO EDPRINT))))
 ; Go command -- boring.
 ((EQUAL COMMAND 'GO)
  (SETQ POV (ED-GO (READ) BODY POV))
  (GO EDPRINT))
 ; Exit -- even more boring.
 ((EQUAL COMMAND 'EXIT)
  (RETURN BODY))
 ; Abort -- same boringness as Exit, I guess.
 ((EQUAL COMMAND 'ABORT)
  (RETURN ()))
 ; If nothing worked out then yell at the user and get a
```

```
        ; new command.
        (T (PRINT ' "ILLEGAL COMMAND.")
           (GO EDREAD))
 )
)))
;
; ED-FIXNUM changes negative position values into their positive
; equivalents. If this can't be done then it screams and give nil.
;
(DEFINE (ED-FIXNUM (LAMBDA (WINDOW N)
 (COND ((OR (ZERO N) (GREATER N (LENGTH WINDOW))
            (GREATER (MULT -1 N) (LENGTH WINDOW)) )
        (PRINT ' "INVALID ELEMENT.")
        NIL)
       ((GREATER 0 N) (ADD 1 (ADD (LENGTH WINDOW) N)))
       (T N)
 )
)))
;
; ED-PRINT does two jobs. It displays the current window in compressed
; form where all lists are replaced by "&" so that long lists can be
; easily read. ED-PRINT also returns the current window [in full] so
; that caller can utilize that information and not have to recalculate.
;
(DEFINE (ED-PRINT (LAMBDA (BODY POV)
        ; If this is the right place then print her and return.
 (COND ((NULL POV) (PRINT (MAPCAR 'ED-TRANS BODY)) BODY)
        ; otherwise standard recursion on the selected subelement.
        (T (ED-PRINT (ED-NTH BODY (CAR POV)) (CDR POV)))
 )
)))
(DEFINE (ED-TRANS (LAMBDA (N)
 (COND ((ATOM N) N)
       (T '&)
 )
)))
;
; ED-POV-PRINT and ED-POV-EXPAND are used to display the lexical name
; of the current POV starting with TOP: .... The only hairiness in this
; is that in order not to clutter the oblist, after the name is made,
; it is REMOBed.
;
(DEFINE (ED-POV-PRINT (LAMBDA (POV)
 (PROG (WORK)
```

```
        (SETQ WORK (PRINT (IMPLODE (CONS ' "TOP:"
                                         (ED-POV-EXPAND POV)))))
        ; Remob the name unless it was () pov in which case you
        ; don't want to remob "TOP:".
        (COND (POV (EVAL (LIST 'REMOB WORK))))
 )
)))
; This guy just put ":"s in between the parts of the POV.
(DEFINE (ED-POV-EXPAND (LAMBDA (POV)
 (COND ((NULL POV) ())
       (T (CONC (LIST (CAR POV) ':)
                (ED-POV-EXPAND (CDR POV)) ))
 )
)))
;
; ED-DETAIL is used to remove the tail from things -- primarily the POV.
;
(DEFINE (ED-DETAIL (LAMBDA (L)
 (COND ((EQUAL 1 (LENGTH L)) ())
       (T (CONS (CAR L) (ED-DETAIL (CDR L))))
 )
)))
;
; ED-INSERT does the task of inserting before/after/ or for an existing
; element in the current window. It is passed the window and three READ
; inputs that indicate the various argument to I in order. The type of
; insert should have been normed by called to one of "B", "A", or "F".
;
(DEFINE (ED-INSERT (LAMBDA (WINDOW WHAT HOW WHERE)
        ; The left hand of this cond selects the proper test according
        ; to the B/A/F option selected.
 (COND ((COND ((EQUAL HOW 'B) (EQUAL WHERE 1))
              ((EQUAL HOW 'A) (EQUAL WHERE 0))
              (T (EQUAL WHERE 1)) )
                ; For "For" kill the selected top element
        (COND ((EQUAL HOW 'F)
               (CONS WHAT (CDR WINDOW)))
               (T (CONS WHAT WINDOW))
        )
       )
        ; Recursion if we haven't yet found the right place then hold
        ; the car element and scan the rest of the list.
        (T CONS (CAR WINDOW)
                 (ED-INSERT (CDR WINDOW) WHAT HOW (SUB WHERE 1))
```

```
          ))
 )
)))
;
; ED-REPLACE is used to put a window into a structure at a particular
; POV point. It breaks the structure up "around" the pov trail, jams
; the new object in place and then zips the structure up again.
;
(DEFINE (ED-REPLACE (LAMBDA (BODY POV NEW)
 (COND ((NULL POV) NEW)
       (T (CONC (ED-LEFTBREAK BODY (CAR POV))
           (CONS (ED-REPLACE (ED-NTH BODY (CAR POV)) (CDR POV) NEW)
                 (ED-RIGHTBREAK BODY (CAR POV))
           )
          )
       )
 )
)))
;
; ED-RIGHTBREAK takes the right of a list beginning with the element
; AFTER the Nth. That is, (ed-rightbreak '(1 2 3) 2) = (3)
;
(DEFINE (ED-RIGHTBREAK (LAMBDA (L N)
 (COND ((ZERO N) L)
       (T (ED-RIGHTBREAK (CDR L) (SUB N 1)))
 )
)))
;
; ED-LEFTBREAK takes the left N elements of a list. That is,
;               (ed-leftbreak '(1 2 3) 2) = (1)
;
(DEFINE (ED-LEFTBREAK (LAMBDA (L N)
 (COND ((EQUAL N 1) ())
       (T (CONS (CAR L) (ED-LEFTBREAK (CDR L) (SUB N 1))))
 )
)))
;
; ED-DELETE takes a location and simply returns the window without the
; specified element.
;
(DEFINE (ED-DELETE (LAMBDA (WINDOW WHERE)
 (COND ((EQUAL 1 WHERE) (CDR WINDOW))
       (T (CONS (CAR WINDOW) (ED-DELETE (CDR WINDOW) (SUB WHERE 1))))
 )
```

```
)))
;
; ED-INSERT-CHECK normalizes the input to insert. It fixes numbers and
; replaces FOR/BEFORE/AFTER for their one letter equivalents.
;
(DEFINE (ED-INSERT-CHECK (LAMBDA (WINDOW WHAT HOW WHERE)
 (COND ((AND (ED-MEMBER HOW '(F FOR A AFTER B BEFORE))
             (SETQ WHERE (ED-FIXNUM WINDOW WHERE)) )
        (ED-INSERT WINDOW WHAT
           (COND ((EQUAL HOW 'FOR) 'F)
                 ((EQUAL HOW 'AFTER) 'A)
                 ((EQUAL HOW 'BEFORE) 'B)
                 (T HOW)
           )
           WHERE
        )
       )
       (T (PRINT ' "ILLEGAL INSERT COMMAND.")
          ())
 )
)))
;
; ED-BACK, ED-GO, and associated routines were added to the editor
; by Stewart Schiffman.
;
; ED-BACK arranges the POV so that the window has shifted treewise left.
; If you hit the front, an error is returned. The tough part of these
; maneuvers is that they have to skip over atomic elements since the
; window cannot contain an atom.
;
(DEFINE (ED-BACK (LAMBDA (POV BODY)
 (PROG (CMD OPOV WINDOW)
  (SETQ OPOV POV)
 BLOOP
  ; Try to move the end pointer in the POV back by one.
  (SETQ CMD (SUB (ED-NTH POV (LENGTH POV)) 1))
  (SETQ POV (ED-DETAIL POV))
  ; Figure out what the window looking down on this one sees.
  (SETQ WINDOW (ED-SETW BODY POV))
  ; Test for hitting the front. If not then fix the POV.
  (COND ((EQUAL CMD 0) (PRINT ' "CANNOT GO BACK.")
                       (SETQ POV OPOV) (RETURN POV)
        )
        (T (SETQ POV (APPEND POV CMD))
```

```
          (SETQ WINDOW (ED-SETW BODY POV)))
  )
  (COND ((ATOM WINDOW) (GO BLOOP)))
  (RETURN POV)
 )
)))
;
; ED-CHNUM and ED-SETW are used by BK and NX to get window images and
; check for illegal position values.
;
(DEFINE (ED-CHNUM (LAMBDA (WINDOW N)
 (NOT (GREATER N (LENGTH WINDOW)))
)))
(DEFINE (ED-SETW (LAMBDA (BODY POV)
 (COND ((ATOM BODY) BODY)
       ((NULL POV) (MAPCAR 'ED-TRANS BODY) BODY)
       (T (ED-SETW (ED-NTH BODY (CAR POV)) (CDR POV)))
 )
)))
;
; ED-NEXT is exactly like ED-BACK except that it moves forward instead
; of backward [obviously].
;
(DEFINE (ED-NEXT (LAMBDA (POV BODY)
 (COND ((NULL POV) (SETQ POV '(3)))
       (T (PROG (CMD OPOV WINDOW)
                (SETQ OPOV POV)
              ILOOP
                (SETQ CMD (ADD (ED-NTH POV (LENGTH POV)) 1))
                (SETQ POV (ED-DETAIL POV))
                (SETQ WINDOW (ED-SETW BODY POV))
                (COND ((NULL (ED-CHNUM WINDOW CMD))
                             (PRINT ' "CANNOT GO FORWARD.")
                             (SETQ POV OPOV) (RETURN POV)
                           )
                           (T (SETQ POV (APPEND POV CMD))
                              (SETQ WINDOW (ED-SETW BODY POV)))
                      )
                      (COND ((ATOM WINDOW) (GO ILOOP)))
                      (RETURN POV)
          )
       )
 )
)))
```

```
;
; ED-GO simply replaces the POV. It has to make sure that the new
; value is not an atom.
;
(DEFINE (ED-GO (LAMBDA (NPOV BODY POV)
 (COND ((ATOM NPOV (PRINT ' "ILLEGAL GO COMMAND: MUST BE A
 LIST.") POV)
          (COND ((ATOM WINDOW) (PRINT ' "CANNOT GO THERE.") POV)
                (T NPOV) ) )
 )
)))
;
; ED-NTH and ED-MEMBER are just utilities that work as one would expect.
; That is, they return the Nth element of a list and find out whether a
; particular target is in a list.
;
(DEFINE (ED-NTH (LAMBDA (L N)
 COND ((EQUAL 1 N) (CAR L))
      (T (ED-NTH (CDR L) (SUB N 1)))
 )
)))
(DEFINE (ED-MEMBER (LAMBDA (A L)
 (COND ((NULL L) ())
       ((EQUAL A (CAR L)) T)
       (T (ED-MEMBER A (CDR L)))
 )
)))
```

Contents | The P-Lisp Interpreter

Contents

Note: Words in CAPITAL letters indicate P-LISP built-ins or functions defined in this book.

" to enclose atoms
& notation in ED
; comment character

ADD function
Algorithm specification
Alphanumeric atom
Alternating saves
AND function
APPLY function
Arguments 1 2 3
Assumption specification
Atom 1 2
ATOM predicate
Atom value 1 2

Balanced parentheses
Binary tree
Binding
Branches
Built-in functions 1 2

CAR function
CDR function
Cells
Collection variable
Comments (in Lisp)
Compiler
CONC function
COND function 1 2
CONS function

EXPLODE function
EXPRs

Factorial 1 2
FEXPRs 1 2
FLAMBDA
Floating point 1 2
Form of call
Formal arguments 1 2
Function definition
Function name 1 2 3 4
Function tracing

Garbage collector
GET function
Global environment
GO function (not in ED)
GREATER predicate

Help functions 1 2

IMPLODE function
INT function
Internal consistency
Interpreter
Interrupt 1 2

LAMBDA 1 2 3
LAMBDA binding
LAMBDA list
Leaves

Phonebook representation
Pig Latin
Pig Latin algorithm
Point of view [POV] in ED
Polynomials
Polynomial representation in Lisp
Predicate 1 2
Pretty printing (see PPRINT in ED)
Problem specifications
PROG 1 2
  function 1 2
  labels
  local variables
Prompts
Properties
Pseudo-code
PUT function

Quote (double)
Quote (single)
QUOTE function

READ routine
Read-Eval-Print 1 2
Recursion 1 2
REM function
RETURN function
Return key
Root

SAVE function

Control structures
Control-C
CxxxxR abbreviations

DEFINE function 1 2 3

ED (P-LISP EDitor) 1 2
& notation
ABORT command
BX command
DELETE command
EXIT command
GO command
INSERT command
level indicator
listing of ED code
number commands
NX command
P command
POV
PPRINT function
windows
ELIZA
Environment 1 2
EQUAL predicate
Errors 1 2 3
EVAL function

Levels
List
List elements
LIST function
Literal atom
LOAD function
Local environments
Local PROG variables
Locals
Loop 1 2

MAPCAR function 1 2 3
Meaning of function names
MEMBER function
MULT function

Nested lists
NIL 1 2 3 4 5 6 7 8
NOT predicate 1 2
NULL predicate 1 2
NUMBER predicate
Numeric atom

OBLIST
OR function

P-LISP interpreter
Page zero

Scope
Self-defining atom
SETQ function 1 2
Side effect
Stack of values
SUB function
Subexpression
SUBR property
Suspended evaluation 1 2
System

T 1 2 3 4
Termination condition 1 2
Time-date stamp saves
Top down programming
Top level elements
TRACE function 1 2 3
Tree

Undefined atom
UNTRACE function

Value
Value stack 1 2
Version saves

ZERO predicate

Contents