Successful Lisp - About

Table of Contents

About this Book........1

Acknowledgments........3

Appendix A - Successful Lisp Applications........5
- Emacs........5
- G2........5
- AutoCAD........6
- Igor Engraver........6
- Yahoo Store........6
- 6

About the Author........7

Chapter 1 - Why Bother? Or: Objections Answered........9
- I looked at Lisp before, and didn't understand it.........9
- I can't see the program for the parentheses.........10
- Lisp is very slow compared to my favorite language.........10
- No one else writes programs in Lisp.........11
- Lisp doesn't let me use graphical interfaces.........11
- I can't call other people's code from Lisp.........12
- Lisp's garbage collector causes unpredictable pauses when my program runs.........12
- Lisp is a huge language.........12
- Lisp is only for artificial intelligence research.........13
- Lisp doesn't have any good programming tools.........13
- Lisp uses too much memory.........13
- Lisp uses too much disk space.........13
- I can't find a good Lisp compiler.........13

Chapter 2 - Is this Book for Me?........15
- The Professional Programmer........15
- The Student........15
- The Hobbyist........16
- The Former Lisp Acquaintance........16
- The Curious........17

Chapter 3 - Essential Lisp in Twelve Lessons........19
- Lesson 1 - Essential Syntax........19

Chapter 3 - Essential Lisp in Twelve Lessons........21
- Lesson 2 - Essential Evaluation........21
  - A form is meant to be evaluated........21
  - A function is applied to its arguments........21
  - A function can return any number of values........23
  - Arguments are usually not modified by a function........23
  - Arguments are usually evaluated before function application........24
  - Arguments are evaluated in left-to-right order........24
  - Special forms and macros change argument evaluation........24

Table of Contents

Chapter 3 - Essential Lisp in Twelve Lessons **25**
- Lesson 3 - Some Examples of Special Forms and Macros 25
  - SETQ 25
  - LET 26
  - COND 28
  - QUOTE 28

Chapter 3 - Essential Lisp in Twelve Lessons **31**
- Lesson 4 - Putting things together, and taking them apart 31
  - CONS 31
  - LIST 31
  - FIRST and REST 32

Chapter 3 - Essential Lisp in Twelve Lessons **35**
- Lesson 5 - Naming and Identity 35
  - A symbol is just a name 35
  - A symbol is always unique 35
  - A symbol can name a value 36
  - A value can have more than one name 36

Chapter 3 - Essential Lisp in Twelve Lessons **39**
- Lesson 6 - Binding versus Assignment 39
  - Binding creates a new place to hold a value 39
  - Bindings have names 39
  - A binding can have different values at the same time 39
  - One binding is innermost 39
  - The program can only access bindings it creates 40
  - Assignment gives an old place a new value 40

Chapter 3 - Essential Lisp in Twelve Lessons **43**
- Lesson 7 - Essential Function Definition 43
  - DEFUN defines named functions 43
  - LAMBDA defines anonymous functions 44

Chapter 3 - Essential Lisp in Twelve Lessons **45**
- Lesson 8 - Essential Macro Definition 45
  - DEFMACRO defines named macros 45
  - Macros return a form, not values 45

Chapter 3 - Essential Lisp in Twelve Lessons **47**
- Lesson 9 - Essential Multiple Values 47
  - Most forms create only one value 47
  - VALUES creates multiple (or no) values 47
  - A few special forms receive multiple values 47
  - Some forms pass along multiple values 48

Table of Contents

Chapter 3 - Essential Lisp in Twelve Lessons **49**
Lesson 10 - A Preview of Other Data Types 49
Lisp almost always does the right thing with numbers 49
Characters give Lisp something to read and write 50
Arrays organize data into tables 52
Vectors are one-dimensional arrays 53
Strings are vectors that contain only characters 54
Symbols are unique, but they have many values 54
Structures let you store related data 55
Type information is apparent at runtime 56
Hash Tables provide quick data access from a lookup key 56
Packages keep names from colliding 57

Chapter 3 - Essential Lisp in Twelve Lessons **61**
Lesson 11 - Essential Input and Output 61
READ accepts Lisp data 61
PRINT writes Lisp data for you and for READ 62
OPEN and CLOSE let you work with files 63
Variations on a PRINT theme 63

Chapter 3 - Essential Lisp in Twelve Lessons **67**
Lesson 12 - Essential Reader Macros 67
The reader turns characters into data 67
Standard reader macros handle built-in data types 67
User programs can define reader macros 67

Chapter 3 - Essential Lisp in Twelve Lessons **69**

Chapter 4 - Mastering the Essentials **71**
Hands-on! The "toploop" 71
Spotting and avoiding common mistakes 72
Defining simple functions 73
Using global variables and constants 77
Defining recursive functions 79
Tail recursion 81
Exercises in naming 83
Lexical binding, and multiple name spaces 84
Reading, writing, and arithmetic 86
Other data types 88
Simple macros 90
Reader macros 92

Chapter 5 - Introducing Iteration **95**
Simple LOOP loops forever... 95
But there's a way out! 95
Use DOTIMES for a counted loop 96
Use DOLIST to process elements of a list 96
DO is tricky, but powerful 97

Table of Contents

Chapter 6 - Deeper into Structures..........99
- Default values let you omit some initializers, sometimes..........99
- Change the way Lisp prints your structures..........99
- Alter the way structures are stored in memory..........101
- Shorten slot accessor names..........101
- Allocate new structures without using keyword arguments..........102
- Define one structure as an extension of another..........102

Chapter 7 - A First Look at Objects as Fancy Structures..........105
- Hierarchies: Classification vs. containment..........105
- Use DEFCLASS to define new objects..........105
- Objects have slots, with more options than structures..........106
- Controlling access to a slot helps keep clients honest..........107
- Override a slot accessor to do things that the client can't..........108
- Define classes with single inheritance for specialization..........109
- Multiple inheritance allows mix-and-match definition..........110
- Options control initialization and provide documentation..........111
- This is only the beginning.............112

Chapter 8 - Lifetime and Visibility..........113
- Everything in Lisp has both lifetime and visibility..........113
- Lifetime: Creation, existence, then destruction..........113
- Visibility: To see and to be seen by..........113
- The technical names: Extent and Scope..........113
- Really easy cases: top-level defining forms..........113
- Scope and extent of parameters and LET variables..........114
- Slightly trickier: special variables..........114

Chapter 9 - Introducing Error Handling and Non-Local Exits..........117
- UNWIND-PROTECT: When it absolutely, positively has to run..........117
- Gracious exits with BLOCK and RETURN-FROM..........118
- Escape from anywhere (but not at any time) with CATCH and THROW..........119
- Making sure files only stay open as long as needed..........120

Chapter 10 - How to Find Your Way Around, Part 1..........121
- APROPOS: I don't remember the name, but I recognize the face..........121
- DESCRIBE: Tell me more about yourself..........122
- INSPECT: Open wide and say "Ah..."..........123
- DOCUMENTATION: I know I wrote that down somewhere..........123

Chapter 11 - Destructive Modification..........125
- Simple assignment is destructive modification..........125
- The risk of assignment..........125
- Changing vs. copying: an issue of efficiency..........127
- Modifying lists with destructive functions..........128
- RPLACA, RPLACD, SETF ...; circularity..........129
- Places vs. values: destructive functions don't always have the desired side-effect..........129
- Contrast e.g. PUSH and DELETE..........130

Table of Contents

Chapter 11 - Destructive Modification
Shared and constant data: Dangers of destructive changes 130

Chapter 12 - Mapping Instead of Iteration 133
MAPCAR, MAPC, and MAPCAN process successive list elements 133
MAPLIST, MAPL, and MAPCON process successive sublists 133
MAP and MAP-INTO work on sequences, not just lists 134
Mapping functions are good for filtering 135
It's better to avoid mapping if you care about efficiency 136
Predicate mapping functions test sequences 136
SOME, EVERY, NOTANY, NOTEVERY 136
REDUCE combines sequence elements 137

Chapter 13 - Still More Things You Can Do with Sequences 139
CONCATENATE: new sequences from old 139
ELT and SUBSEQ get what you want from any sequence (also, COPY-SEQ) 139
REVERSE turns a sequence end-for-end (also, NREVERSE) 140
LENGTH: size counts after all 140
COUNT: when it's what's inside that matters 140
REMOVE, SUBSTITUTE, and other sequence changers 141
DELETE, REMOVE-DUPLICATES, DELETE-DUPLICATES, and NSUBSTITUTE. 142
FILL and REPLACE 142
Locating things in sequences: POSITION, FIND, SEARCH, and MISMATCH 143
SORT and MERGE round out the sequence toolkit 144

Chapter 14 - Can Objects Really Behave Themselves? 145
Generic functions give objects their behaviors 145
The line between methods and objects blurs for multimethods 146
Methods on non-objects? So where does the method live? 146
Generic functions work by dispatching on argument specializers 146
Object inheritance matters after all; finding the applicable method 147
Method combinations offer further choices 148
Nothing is cast in stone; a peek at the metaobject protocol 151

Chapter 15 - Closures 153
Is it a function of the lifetime, or the lifetime of a function? 153
How to spot a free variable, and what to do about it. 153
Using closures to keep private, secure information. 153
Functions that return functions, and how they differ from macros. 154

Chapter 16 - How to Find Your Way Around, Part 2 157
DISASSEMBLE: I always wondered what they put inside those things... 157
BREAK and backtrace: How did I end up here? 159
TRACE and STEP: I'm watching you! 160

Chapter 17 - Not All Comparisons are Equal 163
The longer the test, the more it tells you 163
EQ is true for identical symbols 163

Table of Contents

Chapter 17 - Not All Comparisons are Equal
EQL is also true for identical numbers and characters....163
EQUAL is usually true for things that print the same....163
EQUALP ignores number type and character case....164
Longer tests are slower; know what you're comparing....164
Specialized tests run faster on more restricted data types....164

Chapter 18 - Very Logical, Indeed....167
AND and OR evaluate only as much as they need....167
Bits, bytes, and Boole....167
Bit vectors can go on forever....169
Chunks of bits make bytes....170

Chapter 19 - Streams....173
Streams provide a pipe to supply or accept data....173
Creating streams on files....174
Creating streams on strings....176
Binary I/O....176

Chapter 20 - Macro Etiquette....179
Macros are programs that generate programs....179
Close up: how macros work....179
Backquote looks like a substitution template....180
Beyond the obvious, part 1: compute, then generate....181
Beyond the obvious, part 2: macros that define macros....183
Tricks of the trade: elude capture using GENSYM....184
Macros vs. inlining....186

Chapter 21 - Fancy Tricks with Function and Macro Arguments....189
Keywords let you name your parameters....189
Default values for when you'd rather not say....189
Add some structure to your macros by taking apart arguments....191

Chapter 22 - How to Find Your Way Around, Part 3....195
Class and method browsers help you find your way in a sea of objects....195
ADVISE lets you modify a function's behavior without changing the function....196
WATCH lets you open a window on interesting variables....196

Chapter 23 - To Err is Expected; To Recover, Divine....197
Signal your own errors and impress your users....197
Categorize errors using Conditions....197
Recover from Conditions using Restarts....199

Chapter 24 - FORMAT Speaks a Different Language....207
FORMAT rhymes with FORTRAN, sort of......207
Formatting....207
Iteration....210
Conditionals....210

Table of Contents

Chapter 24 - FORMAT Speaks a Different Language
- Floobydust....211

Chapter 25 - Connecting Lisp to the Real World....213
- Foreign Function Interfaces let you talk to programs written in "foreign languages"....213
- Would you wrap this, please?....213
- I'll call you back.......214
- Network Interfaces: beyond these four walls....214

Chapter 26 - Put on a Happy Face: Interface Builders....215
- Event-driven interfaces....215
- Graphical programming....215
- Example: MCL's Interface Toolkit....215
  - Creating a simple dialog....216
  - Editing a menu....217
- Platform-independent interfaces....218

Chapter 27 - A Good Editor is Worth a Thousand Keystrokes....219
- Simple syntax; smart editors....219
- Matching and flashing....219
- Automatic indentation....219
- Symbol completion....220
- Finding definitions....220
- On-line documentation....220
- Access to debugging tools....220
- Extending the editor using Lisp....221

Chapter 28 - Practical Techniques for Programming....223
- Elements of Lisp style....223
- Property lists are handy for small (very small) ad-hoc databases....223
- Declarations help the compiler, sometimes....224
- DEFVAR versus DEFPARAMETER....225
- Define constants with DEFCONSTANT....226
- Know when (not) to use the compiler....227
- Speed vs. ability to debug....227
- Efficiency: spotting it, testing it....227
- Recognizing inefficiency, profiling; performance vs. readability....229

Chapter 29 - I Thought it was Your Turn to Take Out the Garbage....231
- What is garbage?....231
- Why is garbage collection important?....231
- How does garbage collection work?....231
- What effect does garbage have on my program?....232
- How can I reduce garbage collection pauses in my program?....232

Chapter 30 - Helpful Hints for Debugging and Bug-Proofing....235
- Finding the cause of an error....235
- Reading backtraces, compiler settings for debugging....235

Table of Contents

Chapter 30 - Helpful Hints for Debugging and Bug-Proofing

Simple debugging tools....236
BREAK, PRINT....236
Power tools for tough problems....236
TRACE, STEP, ADVISE, WATCH....236
Into the belly of the beast....237
INSPECT, DESCRIBE....237
Continuing from an error....237
Problems with unwanted definitions....238
Package problems; method definitions....238
The problem with macros....239
Runtime tests catch "can't happen cases" when they do.......239
Use method dispatch rather than case dispatch....239

Chapter 31 - Handling Large Projects in Lisp....241
Packages keep your names separate from my names....241
System builders let you describe dependencies....241
Source control systems keep track of multiple revisions....242
Modules: another way to describe file dependencies....242
PROVIDE and REQUIRE....242

Chapter 32 - Dark Corners and Curiosities....245
Extended LOOP: Another little language....245
TAGBODY: GO if you must....245
Processes & Stack Groups: Juggling multiple tasks....246
Series: Another approach to iteration and filtering....249

Chapter 33 - Where to Go Next....251
Suggestions for further reading....251
Historical References....251
On-line sources....252
Commercial vendors....252

Chapter 34 - Lisp History, or: Origins of Misunderstandings....253
John McCarthy's Notation....253
Earliest implementations....253
Special hardware....253
Diverging dialects....253
The DARPA directive....253
East vs. West, and European competition....253
The emergence of compilers for stock hardware....254
The long road to standardization....254
State of the Art?....254

Table of Contents....257

Table of Contents

Copyright......269

Successful Lisp: How to Understand and Use Common Lisp......271

Successful Lisp: How to Understand and Use Common Lisp......273

Credits......275

Dedication......277

Foreword......279

Introduction......281

Professional Track Suggested Chapter List......283
  Skim for basics:......283
  Read for new concepts:......283
  Learn the environment and tools:......283
  Additional topics:......283

Student Track Suggested Chapter List......285
  Learn the whole language:......285
  Read as interested:......285

Hobbyist Track Suggested Chapter List......287
  Learn how to find your way:......287
  Read as interested:......287

Former User Track Suggested Chapter List......289
  Read as needed:......289

Curious Reader Track Suggested Chapter List......291
  Read in order, skimming as desired:......291

About this Book

This book was produced on Apple Macintosh computers.

I used Digitool's Macintosh Common Lisp to edit and test the book's sample code. Bill St. Clair's HTML-Editor.lisp extended MCL's editor to handle the HTML markup language used in the construction of this book. This way, I was able to edit and test the Lisp code directly in the book's source files and avoid errors I might have otherwise made by cutting and pasting sample code.

I used various Web browsers to view the book.

I used Excalibur by Robert Gottshall and Rick Zaccone to spell check the book, and the computer version of The American Heritage Dictionary Deluxe Edition by Wordstar International for my dictionary and thesaurus.

All software and hardware used in the production of this book was purchased at my own expense. I support shareware products by purchasing those I use.

Contents | Cover
About the Author | About the Book | Dedication

Copyright © 1995-2001, David B. Lamkins
All Rights Reserved Worldwide

This book may not be reproduced without the written consent of its author. Online distribution is restricted to the author's site.

Acknowledgments

Thanks to all of the people in the Lisp community who have made it possible for me to produce this, my first book. Over the years, members of the Usenet newsgroup comp.lang.lisp have provided sage advice, patient tutorials, historical insight, food for thought, and (perhaps unintentionally) amusement. Thanks especially to Barry Margolin, Kent Pitman, Erik Naggum for their contributions to the ongoing dialog, and to Howard Stearns and Mark Kantrowitz for their stewardship of community resources.

Contents | Cover
Copyright | Acknowledgments | Foreword

Copyright © 1995-2001, David B. Lamkins
All Rights Reserved Worldwide

This book may not be reproduced without the written consent of its author. Online distribution is restricted to the author's site.

Appendix A - Successful Lisp Applications

Lisp is used to build applications in which the underlying data is complex and highly dynamic, and where symbolic manipulation is a key feature of the application. Lisp is also used in situations where an application must be readily updated and customized.

You should be aware that Common Lisp is unlikely to become a key component of any commercial operating system in the forseeable future. It is very difficult to merge Lisp storage management techniques with code compiled for traditional pointer-centric languages. Therefore, Lisp must continue to provide its own operating environment. As a Lisp programmer, you should set your expectations accordingly. Once you become facile with Lisp, you'll tend to use Lisp as a workbench and proving ground for algorithmic ideas and programming utilities both large and small. Just don't expect to write and sell a small, simple program in Lisp; the size and cost of the runtime system that you must distribute with your program may be prohibitive.

On the other hand, Lisp excels as a framework for developing large, complex applications. Once past a certain size threshold (for program or data), the fixed overhead of a Lisp environment becomes inconsequential. Also, that rich development environment makes it possible to develop large applications with fewer programmers since much of the algorithmic framework is already provided by Common Lisp's built-in data types and functions. Knowledgeable Lisp programmers achieve even greater productivity by using Common Lisp to write application specific programming tools and language constructs.

The message I want you to take away is that learning and using Lisp is a process. Unless the computer software industry undergoes an unexpected (and unlikely) change, having read this book won't get you a job. However, I'm hoping that what you've learned here will forever color your perceptions as to what is possible, and suggest both techniques and ways of thinking that will be helpful to you in building those "small" applications like word processors, spreadsheets, and OLTP systems.

Meanwhile, I'd like to show you some of the more visible applications that people have developed using Lisp.

- Emacs
- G2
- AutoCAD
- Igor Engraver
- Yahoo Store
- ...

Emacs

More programmers are familiar with Emacs than with any other Lisp application. Richard Stallman conceived Emacs as an extensible editor. He wrote his own Lisp interpreter (not a compiler) specifically for the tasks used in editing text - the low-level text manipulation functions are built-in functions of Emacs Lisp. Over the decades, Emacs has grown to accomodate windowing systems and has accumulated a vast library of code to support programming, writing and personal communications.

G2

Gensym wrote their G2 real-time expert system in Lisp, and later (at greater cost and effort) ported it to C to meet customer expectations. Gensym was able to create a real-time system in Lisp through careful attention to memory allocation; they eliminated unpredictable garbage-collection pauses by simply not generating garbage. (See Chapter 29.)

AutoCAD

AutoCAD is a large-scale Computer Aided Design application. A complex design can contain thousands or millions of parts having complex hierarchical and semantic relationships. Like many application developers who face similar problems, the AutoCAD developers leaned heavily on Lisp for their tools and techniques.

Igor Engraver

Igor Engraver, written by a small team of programmers at Noteheads.com, is an editing and publishing system for musical scores. They've integrated a direct-manipulation graphical interface with a rule-based system that automatically adjusts the layout to conform to complicated rules to meet the expectations of musicians and publishers alike. If that's not enough, Engraver can also play your scores using MIDI instruments. Engraver is targeted to professional composers and musicians, who are encouraged to upload their scores to Noteheads' online e-commerce system for purchase by other musicians. Still want more? Engraver has the slickest software update mechanism you'll find anywhere: select the Check For Patches menu item and Engraver will connect to the Noteheads server, download any patches it needs, and upgrade itself all in a few tens of seconds, without interrupting work in progress.

Yahoo Store

Yahoo Store is one of the best high-profile Lisp success stories of the past few years. Paul Graham and his team, working out of an attic loft, built and operated a server-side e-commerce site builder for hundreds of customers using Common Lisp and generic Linux servers. Paul's venture was so successful that it drew the attention of Yahoo, who saw it as a better tool for their online stores. Paul sold his company to Yahoo for $49 million. An interesting aside is that Paul hired a couple dozen extra programmers during Yahoo's due diligence investigations, since "no one would believe that three guys in a loft" could have done what Paul's team accomplished with the help of Lisp. Perception is everything...

...

Each vendor has its own list of success stories. See Chapter 33 for some starting points.

Contents | Cover
Chapter 34 | Appendix A

Copyright © 1995-2001, David B. Lamkins
All Rights Reserved Worldwide

This book may not be reproduced without the written consent of its author. Online distribution is restricted to the author's site.

About the Author

David Lamkins was born in Watervliet, New York. Very little is known about his childhood except that he started taking things apart as soon as he could hold a screwdriver. It wasn't until David reached the age of twelve that he started putting things back together.

This fascination with the inner workings of things carried David forward through a long period of his life (euphemistically dubbed "The Quiet Years"), during which he masterfully combined the roles of computer geek and responsible adult. Of course, this resulted in a rather mundane existence and bored the hell out of everyone around him...

David has recently rediscovered that people are more fun to play with than symbols, and has decided to -- in the grand tradition of reformed geeks everywhere -- get a life.

After thirty years of half-heartedly messing around on the guitar (including some stints in very short-lived bands during his teen years) David has finally decided that he'd like to be a musician and play in a band that performs dark, langorous, fluid music; an alchemic transmutation of psychedelia and metal.

David's favorite movies are "The Fifth Element" and "The Matrix".

Contents | Cover
About the Author | About the Book

Copyright © 1995-2001, David B. Lamkins
All Rights Reserved Worldwide

This book may not be reproduced without the written consent of its author. Online distribution is restricted to the author's site.

Chapter 1 - Why Bother? Or: Objections Answered

Everyone knows Lisp, right? Many of us took a course that introduced us to Lisp along with three or four or more other languages. This is how I was introduced to Lisp around 1975, and I thought it was a pretty useless language. It didn't do *anything* in the usual way, it was slow, and those parentheses were enough to drive anyone crazy!

If your own Lisp experience predates 1985 or so, you probably share this view. But in 1984, the year Big Brother never *really* became a reality (did it?), the year that the first bleeding-edge (but pathetic by today's standards) Macintosh started volume shipments, the Lisp world started changing. Unfortunately, most programmers never noticed; Lisp's fortune was tied to AI, which was undergoing a precipitous decline -- The AI Winter -- just as Lisp was coming of age. Some say this was bad luck for Lisp. I look at the resurgence of interest in other dynamic languages and the problems wrestled with by practicioners and vendors alike, and wonder whether Lisp wasn't too far ahead of its time.

I changed my opinion of Lisp over the years, to the point where it's not only my favorite progamming language, but also a way of structuring much of my thinking about programming. I hope that this book will convey my enthusiasm, and perhaps change your opinion of Lisp.

Below I've listed most of the common objections to Lisp. These come from coworkers, acquaintances, managers, and my own past experience. For each point, I'll describe how much is actually true, how much is a matter of viewpoint, and how much is a holdover from the dark days of early Lisp implementations. As much as possible, I'll avoid drawing comparisons to other languages. Lisp has its own way, and you'll be able to make your own comparisons once you understand Lisp as well as your usual language. If you eventually understand Lisp enough to know when its use is appropriate, or find a place for Lisp in your personal toolkit, then I've done my job.

Without further introduction, here are a baker's dozen reasons why you might be avoiding Lisp:

I looked at Lisp before, and didn't understand it.

This is a really tough one. Most programming languages are more similar to each other than they are to Lisp. If you look at a family tree of computer languages, you'll see that the most common languages in use today are descendants of the Algol family. Features common to languages in the Algol family include algebraic notation for expressions, a block structure to control visibility of variables, and a way to call subroutines for value or effect. Once you understand these concepts, you can get started with another language in the family by studying the surface differences: the names of keywords and the style of punctuation.

Lisp really *is* different. If you've only read code in Algol family languages, you'll find no familiar punctuation or block structure to aid your understanding of Lisp code -- just unfamiliar names appearing in seemingly pointless nests of parentheses. In Lisp, the parenthesis *is* the punctuation. Fortunately, its use is quite simple; simpler than, for example, remembering the operator precedence rules of C or Pascal. Lisp development environments even provide editors that help with matching opening and closing parentheses.

Once you understand how Lisp expressions are put together, you still have to learn what they mean. This is harder because Lisp provides a lot of facilities that aren't found elsewhere, or gives unfamiliar names to familiar concepts. To really understand Lisp, you need to know how it works inside. Like most good programmers, you probably have a mental model of how a computer works, and how your favorite compiler translates statements from your favorite language into machine code. You'll drive yourself crazy if you try this with Lisp, which seems to go to great lengths to isolate you from the details of machine organization. Yes, you sacrifice some control. Perhaps not surprisingly, you gain quite a lot in program correctness once you give up worrying about how your program is mapped by the compiler onto bits in the machine. Is the tradeoff worthwhile? We'll explore that issue in a later chapter.

This book will teach you how to read and write Lisp, how to recognize and understand new words like `DEFUN`, `CONS`, and `FLET`, and -- ultimately -- how to think in Lisp as well as you think in your favorite programming language.

I can't see the program for the parentheses.

Part of this problem is a matter of dealing with the unfamiliar. I talked about that in the previous section. Another part of this problem is real: you have to deal with a lot of parentheses. Fortunately, Lisp programming environments have editors that mechanize the process of counting parentheses by flashing or highlighting matching pairs or by manipulating entire balanced expressions. Finally, there's a matter of style. Judicious indentation improves the readability of Lisp programs, as it does in other languages. But vertical whitespace often hinders readability in Lisp.

I'll cover both the mechanical and stylistic aspects of Lisp code in this book. By the time you're done, you'll have an opinion on what constitues readable code, and you'll be able to defend your position. When you reach that level of confidence, you'll be able to write aesthetic Lisp code, and to read anyone else's code. Parentheses won't be a concern any longer.

Lisp is very slow compared to my favorite language.

Possibly... But the difference may not be as large as you'd expect. First, let's clear the table of an *old* misconception: that Lisp is an interpreted language. As a rule, most modern Lisp systems compile to machine code. A few compile to byte code that typically runs five times slower than machine code. And one or two freeware Lisp systems only run interpreted code, but they're the exception. So there's part one of the answer: if you're not running a Lisp *compiler*, you should get one.

Your Lisp coding style affects execution speed. Unfortunately, you won't recognize inefficient Lisp code until you've had some experience with the language. You'll need to think about how Lisp works in order to understand what makes Lisp code run slowly. This is not hard to do, but the issues are different from those for languages which expose more of the underlying machine to you.

Lisp gives you incremental compilation. This means that you can compile one function at a time and be ready to run your program instantly -- there is no linkage step. This means that you can make lots of changes quickly and evaluate them for their effect on the program. Lisp also has built-in instrumentation to help you tune the performance of your program.

You'll experience all of these things as you work your way through this book. By the time you're done, you'll know how to avoid writing inefficient code in the first place, and how to use all of the available tools to identify and fine tune the really critical code in your programs.

No one else writes programs in Lisp.

What? I'm the only one left? I don't think so...

Seriously, though, there are quite a few people who write Lisp code every day. They write programs that solve tough problems, and give their employers a strategic advantage. It's hard to find good Lisp programmers who are willing to move to a new employer; those companies who are using Lisp guard their strategic advantage, and their Lisp programmers, quite jealously.

Now, it's mostly true that you won't find Lisp in consumer products like spreadsheets, databases, word processors, and games. But then, that's not the kind of work that Lisp does best. You *will* find Lisp in products that must reason about and control complex systems and processes, where the ability to reliably arrive at useful conclusions based upon complex relationships among multiple sources and kinds of data is more important than lightning-fast numerical calculations or spiffy graphics (although modern Lisp systems come pretty close to the leaders even in the latter two categories).

Lisp is also used as an extension language because of its simple, consistent syntax and the ability for system designers to add new functions to Lisp without writing an entire new language. The Emacs editor and the AutoCAD drafting program are two of the best examples of this use of Lisp.

And of course Lisp is *still* the language most often used for research in artificial intelligence and advanced computer language design, but we won't touch on either of those subjects in this book. When you've finished this book, you'll have the knowledge needed to recognize what problems you should solve using Lisp, and how to approach the solution's design.

Oh, and one more thing: It's not quite true that no mass market product uses Lisp. Microsoft's "Bob" environment for naive computer users was developed (and delivered) in Lisp.

Lisp doesn't let me use graphical interfaces.

This is ironic. Some of the first graphical user interfaces appeared on Lisp machines in the early 1970s. In fact, in 1995 you can still buy a DOS adaptation of one of these early Lisp environments -- with the same GUI it had twenty years ago.

The leading Lisp development environments for Windows and Macintosh support only a subset of their host platform's GUI. It's possible to add support for the missing features, but easier to do it using Microsoft's and Apple's preferred language: C++.

If you want to have the same graphical user interface on your Lisp program when it runs on Windows or Macintosh hosts, you can find at least two Lisp windowing environments that let you do this. The problem is that the Lisp GUI will be familiar to neither Macintosh nor Windows users.

If all you want is a quick, platform-specific graphical interface for your Lisp program, any of the commercial Lisp environments will deliver what you need. They all have graphical interface builders that let you build windows and dialogs with point and click or drag and drop techniques. Just don't expect much in the way of bells and whistles.

I can't call other people's code from Lisp.

This is mostly untrue. Most Lisp environments give you a way to call external routines using either C or Pascal calling conventions. You can also call back into Lisp from the external program. But if you want to call C++ from Lisp, you'll probably have to write a C wrapper around the C++ code.

Lisp's garbage collector causes unpredictable pauses when my program runs.

This should probably be covered in the "Lisp is slow" discussion, but there are enough interesting digressions for this to warrant its own topic. Lisp programs create garbage by destroying all references to some object in memory. In a program written in some other language, the programmer must arrange to release the memory occupied by the object at the same time when the last reference is destroyed. If the program fails to do this reliably, the program has a *memory leak* -- eventually the program's memory space could fill up with these unreachable objects and not leave enough free memory for the program to continue. If you've ever written a complex program that allocates and manually recycles a lot of dynamic memory, you know how difficult a problem this can be.

Lisp finesses the memory leakage problem by *never* allowing the programmer to release unused memory. The idea here is that the computer can determine when a block of memory is unreachable with complete accuracy. This unreachable block is said to be garbage because it is no longer useful to any part of the program. The garbage collector runs automatically to gather all these unused blocks of memory and prepare them for reuse. The algorithms that do this are very tricky, but they come built into your Lisp system.

Historically, garbage collection *has been* slow. The earliest garbage collectors could literally lock up a system for hours. Performance was so poor that early Lisp programmers would run with garbage collection turned off until they completely ran out of memory, then start the garbage collection manually and go home for the rest of the day.

Over the past twenty years, a lot of good software engineering techniques have been applied to improving the performance of garbage collectors. Modern Lisp systems collect garbage almost continuously, a little bit at a time, rather than waiting and doing it all at once. The result is that even on a very slow desktop machine a pause for garbage collection will rarely exceed a second or two in duration.

Later in this book I'll discuss garbage collection in greater detail and show you techniques to avoid generating garbage; the less garbage your program creates, the less work the garbage collector will have to do.

Lisp is a huge language.

If you look at the book *Common Lisp: The Language*, weighing in at about a thousand pages, or the recent (and bulkier) *ANSI Standard X3.226: Programming Language Common Lisp*, it's easy to form that opinion. When you consider that the Lisp language has almost *no* syntax, and only a couple of dozen primitive language elements (called special forms), then Lisp starts to look like a *very small* language.

In fact, the manuals cited above are mostly concerned with descriptions of what most other languages would call *library* functions and, to a lesser degree, development tools. Take the language manual for your favorite language. Add the manuals for three or four third-party libraries -- development utilities, fancy data structures, generalized I/O, etc. Take all the manuals for your development tools -- browsers, inspectors, debuggers, etc. and toss them onto the growing pile. Now count the pages. Does a thousand pages still seem like a lot?

By the time you've finished this book, you'll know how to find what you need in Lisp, with or without a manual.

Lisp is only for artificial intelligence research.

Just not true. Lisp gets used for big projects that have to be tackled by one or a few programmers. Lisp is also good for tasks that are not well defined, or that require some experimentation to find the proper solution. As it turns out, artificial intelligence meets all of these criteria. So do a lot of other applications: shop job scheduling, transportation routing, military logistics, sonar and seismological echo feature extraction, currency trading, computer and computer network configuration, industrial process diagnosis, and more. These aren't mass market applications, but they still make lots of money (often by avoiding cost) for the organizations that develop them.

Lisp doesn't have any good programming tools.

I hope to convince you otherwise. Several chapters of this book are devoted to introducing you to the many useful tools provided by a Lisp development environment.

Lisp uses too much memory.

The Lisp development systems on both my Mac and my PC run comfortably in anywhere from 4 to 8 megabytes of RAM. Less in a pinch. The integrated C++ development environments take anywhere from 12 to 20 megabytes. Both have comparable tools and facilities.

Lisp uses too much disk space.

The Lisp development systems on both my Mac and my PC use considerably less disk space than the C++ environments. Lisp space on my hard disk runs from a low of about 5 megabytes for one system to a high of about 30 megabytes for another system that is a total programming environment, including a built in file manager, WYSIWYG word processor, graphics program, appointment calendar, and (almost forgot) Lisp development environment. The C++ systems run from a low of about 20 megabytes to a high of about 150 megabytes.

I can't find a good Lisp compiler.

Depending on what kind of computer you use, this was a problem as recently as a year or two ago. And it's true that there isn't a lot of competition for the Lisp marketplace -- you can count vendors on the fingers of one hand. The vendors who support the Lisp marketplace tend to have been around for a long time and have good reputations. As desktop computers increase in speed and storage capacity, Lisp vendors are increasingly turning their attention to these platforms.

Contents | Cover
Introduction | Chapter 1 | Chapter 2

Copyright © 1995-2001, David B. Lamkins
All Rights Reserved Worldwide

This book may not be reproduced without the written consent of its author. Online distribution is restricted to the author's site.

Chapter 2 - Is this Book for Me?

Depending upon your background, interest, and experience, your need for the information offered in this book is best met by following the material in a certain way. I think that most readers will place themselves in one of the following categories: professional programmer, student, hobbyist, former Lisp user, or merely curious. No matter which category you fit, I've described what I think you can gain by reading this book. As you read through this book, you may decide that you no longer fit your original category. This is fine -- there are no strong dependencies between chapters. If you get stuck on a particular concept, the detailed Table of Contents will help you locate the information you need.

The Professional Programmer

This book tells you what you need to know about Lisp in order to write good programs, and to read Lisp code in journals, magazines, or other people's programs. Beyond that, I will introduce you to some important concepts that you may not have encountered in your use of other languages, or you may find that the Lisp approach to a familiar concept gives you a new perspective on an old idea. Even if you never have occasion to use Lisp on the job, the concepts you'll learn in this book may give you a fresh insight to help solve a tough problem in your favorite language. You'll probably want to skim up through Chapter 6 to make sure you've covered the basics. Then slow down and take a closer look at what interests you in Chapter 7 through Chapter 9, Chapter 11 through Chapter 15, Chapter 17 through Chapter 21, Chapter 23, and Chapter 24; these chapters cover concepts that are either unique to, or best expressed in, the Lisp language.

Beyond all else, I hope to impress upon you the dynamic nature of Lisp program development. Lisp usually is a pleasant surprise to someone accustomed (or resigned) to the usual edit, compile, link, and debug cycle. The biggest change is compilation of functions rather than files. You can change and recompile just one function at a time, even from within the debugger. This is really handy if you've spent hours of testing to find a problem that can be easily fixed with one small change to your program. This is just one example of how the Lisp programming environment supports your programming efforts. You'll find additional examples throughout this book. Chapter 10, Chapter 16, Chapter 22, and Chapter 26 through Chapter 28 will give you an appreciation of how Lisp supports dynamic program development.

Professional Track

The Student

If you've learned Lisp in a typical classroom setting, you may have come to believe that the language is nothing but lists and recursion. This book will show you that Lisp has a rich assortment of data types and control structures. Lists and recursion are only the tip of the iceberg. Chapter 3 through Chapter 24 should fill in the details on the rest of Lisp. Skim the remaining chapters so you know where to look when you have access to a commercial Lisp development environment, for when you begin your first Lisp project outside of an academic setting.

Depending upon whether you're currently taking a Lisp course, or have already finished a course and want to learn what the instructor didn't have time for, this book will help your studies by teaching you an appreciation for the language and the skills you'll need for its most effective use. Appendix A lists sources of Lisp development environments. You can use the freeware versions while you're still a poor, starving student, and refer to the list again when you're gainfully employed and want to either recommend a commercial implementation of Lisp to your employer or buy one for personal use.

Student Track

The Hobbyist

To me, a hobbyist is someone who pursues programming for the challenge, for the learning experience, or as a pastime. The hobbyist is largely self taught. If you fit that mold, I'll warn you now that Lisp can be very challenging, and can teach you a lot about programming.

You can go quite a long way toward learning Lisp with one of the freeware systems available for Macintosh and DOS computers. But if you have aspirations to turn your hobby into a money making venture, you need to ask yourself whether Lisp is appropriate for your anticipated product or service. If you think in terms of databases or scripts or multimedia, you'll probably be happier with a tool that directly addresses your area of interest. If you have dreams of writing the next great videogame, you've probably already discovered that you need a language that lets you program "close to the machine" -- If so, Lisp will disappoint you. But if you want to give your game characters complex interactions, or even the appearance of intelligent behavior, Lisp is a wonderful vehicle in which to design and test prototypes of these behaviors.

No matter what your interest in programming as a hobby, this book will give you the understanding you need to explore Lisp without getting bogged down in the parentheses. Read through all of the chapters, spending more time on those which interest you the most. If you have access to a Lisp development system, spend time on Chapter 10, Chapter 16, Chapter 22, and Chapter 28 through Chapter 30; these chapters will give you the background you need in order to find your way when you get lost -- you'll find this more helpful than trying to develop an encyclopedic knowledge of the language.

Hobbyist Track

The Former Lisp Acquaintance

If you've had a prior experience with Lisp, perhaps in a college or university programming class, this book will update your knowledge. This book will teach you things that a one semester class could never cover due to time constraints. You'll also see how commercial Lisp development systems provide tools and features missing from the freeware Lisp system that your educational institution probably used.

If you've worked on (or have attempted) a Lisp project before, you may not have had the benefit of a mentor to show you how to use Lisp effectively. This book will introduce you to the skills that you need to become a successful Lisp programmer. It is important that you understand what the language does; this book, like others before it, covers that ground. But this book goes beyond other Lisp programming books to tell you why Lisp works as it does, the best way to do things in Lisp, and (perhaps most importantly) how to approach the Lisp development *environment* to accomplish your goals in the most effective way.

I suggest that you use this book as a reference. The detailed Table of Contents will help you find subject areas that appeal to your interests or needs.

Former User Track

The Curious

If you have no immediate intentions of writing a Lisp program (perhaps you're a student of programming languages), this book is still a good choice. You can learn a lot about Lisp, its development environment, and its use by reading through the chapters in order and working out an occasional bit of code on paper, to check your understanding. I've tried hard to introduce the fundamentals of Lisp in a way that doesn't belabor the details of internal representation.

Curious Reader Track

Contents | Cover
Chapter 1 | Chapter 2 | Chapter 3

Copyright © 1995-2001, David B. Lamkins
All Rights Reserved Worldwide

This book may not be reproduced without the written consent of its author. Online distribution is restricted to the author's site.

Chapter 3 - Essential Lisp in Twelve Lessons

Lesson 1 - Essential Syntax

- Lists are surrounded by parentheses

This is the first thing you need to know about Lisp: anything surrounded by parentheses is a list. Here are some examples of things that are lists:

```
(1 2 3 4 5)
(a b c)
(cat 77 dog 89)
```

As I said, anything surrounded by parentheses is a list. When you hear a statement like that, you probably want to ask two questions:

1. What if I put parentheses around *nothing*?
2. What if I put parentheses around *another list*?

In both cases the answer is the same. You still have a list. So the following are also lists:

```
()
(())
((()))
((a b c))
((1 2) 3 4)
(mouse (monitor 512 342) (keyboard US))
(defun factorial (x) (if (eql x 0) 1 (* x (factorial (- x 1)))))
```

The only time you don't have a list is when you have a right parenthesis without a matching left parenthesis or vice versa, as in the following four examples:

```
(a b c(
((25 g) 34
((())
(()))
```

This is nothing to lose sleep over -- Lisp will tell you when there's a mismatch. Also, the editor that you use for writing Lisp programs will almost certainly give you a way to automatically find matching parentheses. We'll look at editors in Chapter 27.

A list can be a lot of things in Lisp. In the most general sense, a list can be either a program or data. And because lists can themselves be made of other lists, you can have arbitrary combinations of data and programs mixed at different levels of list structure -- this is what makes Lisp so flexible for those who understand it, and so confusing for those who don't. We'll work hard to remove that confusion as this chapter continues.

Atoms are separated by whitespace or parentheses Now that you can recognize a list, you'd like to have a name for the things that appear between the parentheses -- the things that are not themselves lists, but rather (in our examples so far) words and numbers. These things are called atoms.

Accordingly, these words and numbers are all atoms:

```
1
25
342
mouse
factorial
x
```

Lisp lets us use just about any characters to make up an atom. For now, we'll say that any sequence of letters, digits, and punctuation characters is an atom if it is preceded and followed by a blank space (this includes the beginning or end of a line) or a parenthesis. The following are all atoms:

```
-
*
@comport
funny%stuff
9^
case-2
```

One thing you should remember, if you're experienced in another programming language, is that characters traditionally reserved as operators have no special meaning when they appear within a Lisp atom. For example, `case-2` is an atom, and *not* an arithmetic expression.

Since an atom can be marked off by either whitespace or a parenthesis, we could eliminate any whitespace between an atom and a parenthesis, or between two parentheses. Thus, the following two lists are identical:

```
(defun factorial (x) (if (eql x 0) 1 (* x (factorial (- x 1)))))
(defun factorial(x)(if(eql x 0)1(* x(factorial(- x 1)))))
```

In practice, we'd *never* write the second list. In fact, we'd probably split the list across multiple lines and indent each line to improve readability; this list is in fact a small program, and formatting it as follows makes it easier to read for a Lisp programmer:

```
(defun factorial (x)
  (if (eql x 0)
    1
    (* x (factorial (- x 1)))))
```

For now, you don't need to worry about what this means or how you'd know to do this kind of indentation. As we get further into this chapter, you'll see more examples of indentation. Subsequent chapters will show additional examples, and I'll point out how to use indentation to improve readability of many new constructs. Chapter 28 will address elements of Lisp style, including the proper use of indentation.

Contents | Cover
Chapter 2 | Chapter 3, Introduction | Chapter 3, Lesson 1 | Chapter 3, Lesson 2 | Chapter 4

Copyright © 1995-2001, David B. Lamkins
All Rights Reserved Worldwide

This book may not be reproduced without the written consent of its author. Online distribution is restricted to the author's site.

Chapter 3 - Essential Lisp in Twelve Lessons

Lesson 2 - Essential Evaluation

A form is meant to be evaluated

A form can be either an atom or a list. The important thing is that the form is meant to be evaluated. Evaluation has a fairly technical definition that we'll gradually expose in this section.

Evaluation is simple if the form is an atom. Lisp treats the atom as a name, and retrieves the value for the name (if a value exists). You probably wonder why I'm avoiding the more direct explanation of calling the atom a variable. The reason is that the atom can have either a variable value or a constant value. And the atom's value can be constant for a couple of reasons.

A number is an atom. (Its value is constant for obvious reasons.) Lisp does not *store* a value for a number -- the number is said to be self-evaluating.

> We're going to introduce a new term without a complete definition. For now, think of a *symbol* as an atom that can have a value. We'll look at symbols in greater detail when we get to Lesson 5.

A symbol defined in a `defconstant` form has a constant value. Lisp will store the value as if the atom had a variable value, and add a note to the effect that the value is not allowed to change.

A symbol in the `KEYWORD` package is self-evaluating. We'll look at packages in detail in Chapter 31. For now, all you need to know is that a symbol beginning with the `:` character (called the package prefix) is a keyword symbol. Keyword symbols have themselves as their values.

A symbol can get a variable value in many different ways. Lisp actually keeps several different values for a symbol. One has the traditional meaning as the value of the symbol taken as a variable. Another has meaning as the symbol's function. Still others keep track of the symbol's documentation, its printed representation, and properties that the programmer chooses to associate with the symbol. We'll explore some of these in more detail in Lesson 5, Lesson 6, and Lesson 7.

If a form is a list, then the first element must be either a symbol or a special form called a lambda expression. (We won't look at lambda expressions for a while.) The symbol must name a function. In Lisp, the *symbols* `+`, `-`, `*`, and `/` name the four common arithmetic operations: addition, subtraction, multiplication, and division. Each of these symbols has an associated function that performs the arithmetic operation.

So when Lisp evaluates the form `(+ 2 3)`, it applies the function for addition to the arguments `2` and `3`, giving the expected result `5`. Notice how the function symbol, `+`, precedes its arguments. This is *prefix* notation. Any time you see a list, look to its first element to find out what Lisp will do to evaluate the list as a form.

A function is applied to its arguments

Lisp, when given a list to evaluate, treats the form as a function call. We'll be looking a lot at Lisp evaluation from now on, so we'll use some visual aids to identify the input to Lisp and its responses:

```
⊙ the Lisp prompt precedes input to Lisp
→ result of Lisp evaluation
```

For example:

```
⊙ (+ 4 9)
→ 13

⊙ (- 5 7)
→ -2

⊙ (* 3 9)
→ 27

⊙ (/ 15.0 2)
→ 7.5
```

In each case above, the evaluated form is a list. Its first element is a symbol, which names a function. The remaining elements are *arguments* of the function. Here, the arguments are all numbers, and we know that numbers are self-evaluating.

Here are a few more examples:

```
⊙ (atom 123)
→ T

⊙ (numberp 123)
→ T

⊙ (atom :foo)
→ T

⊙ (numberp :foo)
→ NIL
```

`ATOM` and `NUMBERP` are predicates. Predicates return a true or false value. `NIL` is the only false value in Lisp -- everything else is true. Unless a predicate has a more useful value to return, it conventionally returns `T` to mean true. `ATOM` returns `T` if its one argument is a Lisp atom. `NUMBERP` returns `T` if its argument is a number.

To evaluate each of the above forms, Lisp first evaluates the arguments (from left to right), then evaluates the first element to get its function, then applies the function to the arguments. With only a handful of exceptions, which we'll learn about at the end of this lesson, Lisp always does the same thing to evaluate a list form:

1. Evaluate the arguments, from left to right.
2. Get the function associated with the first element.
3. Apply the function to the arguments.

Remember that an atom can also be a Lisp form. When given an atom to evaluate, Lisp simply returns its value:

```
⊙ 17.95
→ 17.95

⊙ :A-KEYWORD
→ :A-KEYWORD
```

```
⊙ *FEATURES*
→ (:ANSI-CL :CLOS :COMMON-LISP)

⊙ "Hello, world!"
→ "Hello, world!"

⊙ WHAT-IS-THIS?
⇥ Error: Unbound variable
```

Numbers and keywords are self-evaluating. So are strings. The `*FEATURES*` variable is predefined by Lisp -- your system will probably return a different value.

The symbol `WHAT-IS-THIS?` doesn't have a value, because it's not predefined by Lisp, and I haven't given it a value. The system responds with an error message, rather than a value. We mark the message with ⇥ rather than the → marker we use for successful evaluations. Your system will probably print a different message.

A function can return any number of values

Sometimes you'd like to have a function return several values. For example, a function which looks up a database entry might return both the desired result and a completion status code. One way to do this is to pass to the function a location for one of the results; this is possible, but **very** uncommon for a Lisp program.

Another approach creates a single return value to combine both the result and the status code. Lisp gives you several different ways to do this, including structures. Experienced Lisp programmers don't do this when the created value will just be taken apart into its components and then forgotten, since the composite value then becomes garbage (see Chapter 29) that eventually slows down the operation of the program.

The right way to return multiple values from a function is to use the `VALUES` form. We'll see `VALUES` used in the context of a function in a little while. For now, let's see what happens when Lisp evaluates a `VALUES` form:

```
⊙ (values 1 2 3 :hi "Hello")
→ 1
→ 2
→ 3
→ :HI
→ "Hello"
```

Notice how Lisp returned a value (following the → indicator) for *each* argument to the `VALUES` form. My Lisp system represents this by printing each value on a new line; yours may separate the values some other way.

Arguments are usually not modified by a function

I mentioned earlier that you can pass a location to a function, and have the function change the location's value. This is a *very* uncommon practice for a Lisp program, even though other languages make it part of their standard repertoire.

You could specify the location to be modified as either a non-keyword symbol or a composite value -- obviously, you can't modify a constant. If you provide a symbol, then your function must execute code to give the symbol a new value. If you provide a composite data structure, your function must execute code to change the correct piece of the composite value. It's harder to write Lisp code to do this, and it's harder to understand programs written this way. So Lisp programmers usually write functions that get their inputs from parameters, and produce their outputs as the function result.

Arguments are usually evaluated before function application

When Lisp evaluates a function, it always evaluates all the arguments first, as we saw earlier. Unfortunately, every rule has exceptions, and this rule is no exception (as we'll soon see)... The problem is not that Lisp doesn't always evaluate a function's arguments, but that not every list form is a function call.

Arguments are evaluated in left-to-right order

When a list form *is* a function call, its arguments are always evaluated in order, from left to right. As in other programming languages, it's in poor taste to rely on this, but if you absolutely have to rely on the order, it's good to know that Lisp defines it for you.

Special forms and macros change argument evaluation

So if a list form isn't always a function call, what else can it be? There are two cases, but the result is the same: some arguments are evaluated, and some aren't. Which is which depends upon the form and nothing else. You'll just have to learn the exceptions. Fortunately, most Lisp systems will show you the online documentation for any form with just a keystroke or two.

There are two kinds of forms that don't evaluate all of their arguments: special forms and macros. Lisp predefines a small number of special forms. You can't add your own special forms -- they're primitive features of the language itself. Lisp also defines quite a few macros. You can also define your own macros. Macros in Lisp let you use the full power of the language to add your own features. Later in this chapter we'll look briefly at how to define simple macros. In Chapter 20 we'll cover topics surrounding the creation of more complex macros.

Contents | Cover
Chapter 2 | Chapter 3, Introduction | Chapter 3, Lesson 1 | Chapter 3, Lesson 2 | Chapter 3, Lesson 3 | Chapter 4

Copyright © 1995-2001, David B. Lamkins
All Rights Reserved Worldwide

This book may not be reproduced without the written consent of its author. Online distribution is restricted to the author's site.

Chapter 3 - Essential Lisp in Twelve Lessons

Lesson 3 - Some Examples of Special Forms and Macros

Now we'll look at several special forms and macros. Over the next four lessons, we'll build up a repertoire that will let you write simple functions using the most elementary Lisp data type, the list. Later chapters will cover more complex program structures and data types.

SETQ

Earlier, I told you that Lisp evaluates a symbol form by retrieving its variable value. `SETQ` gives you a way to set that value:

```
⊝ (setq my-name "David")
→ "David"

⊝ my-name
→ "David"

⊝ (setq a-variable 57)
→ 57

⊝ a-variable
→ 57

⊝ (setq a-variable :a-keyword)
→ :A-KEYWORD
```

`SETQ`'s first argument is a symbol. This is not evaluated. The second argument is assigned as the variable's value. `SETQ` returns the value of its last argument.

> `SETQ` doesn't evaluate its first argument because you want to assign a value to the symbol itself. If `SETQ` evaluated its first argument, the *value* of that argument would have to be a symbol. The `SET` form does this. Pay particular attention to the difference between the `SET` and `SETQ` forms in the following example, and make sure you understand what's happening with the `(set var-1 99)` form:
>
> ```
> ⊝ (setq var-1 'var-2)
> → VAR-2
>
> ⊝ var-1
> → VAR-2
>
> ⊝ var-2
> ⇥ Error: Unbound variable
>
> ⊝ (set var-1 99)
> → 99
>
> ⊝ var-1
> → VAR-2
>
> ⊝ VAR-2
> → 99
> ```

> Did you notice the `'` in the first form? This keeps the following form, `var-2`, from being evaluated. Later in this lesson, when we look at `QUOTE`, I'll explain this in greater detail.
>
> In the example, we first make the value of `VAR-1` be the *symbol* `VAR-2`. Checking the value of `VAR-2`, we find that it has none. Next, we use `SET` (not `SETQ`) to assign the value `99` to the symbol, `VAR-2`, which is the *value* of `VAR-1`.

The `SETQ` form can actually take any even number of arguments, which should be alternating symbols and values:

```
⊙ (setq month "June" day 8 year 1954)
→ 1954

⊙ month
→ "June"

⊙ day
→ 8

⊙ year
→ 1954
```

`SETQ` performs the assignments from left to right, and returns the rightmost value.

LET

The `LET` form looks a little more complicated than what we've seen so far. The `LET` form uses nested lists, but because it's a special form, only certain elements get evaluated.

```
⊙ (let ((a 3)
        (b 4)
        (c 5))
    (* (+ a b) c))
→ 35

⊙ a
⇥ Error: Unbound variable

⊙ b
⇥ Error: Unbound variable

⊙ c
⇥ Error: Unbound variable
```

The above `LET` form defines values for the symbols `A`, `B`, and `C`, then uses these as variables in an arithmetic calculation. The calculation's result is also the result of the `LET` form. Note that none of the variables defined in the `LET` have a value after Lisp has finished evaluating the form.

In general, `LET` looks like this:

`(let (`*bindings*`)` *forms*`)`

where *bindings* is any number of two-element lists -- each list containing a symbol and a value -- and *forms* is any number of Lisp forms. The forms are evaluated, in order, using the values established by the bindings. `LET` returns the value(s) returned by the last form.

Indentation doesn't affect the operation of `LET`, but proper indentation does improve readability. Consider these equivalent forms:

```
(let ((p 52.8)
      (q 35.9)
      (r (f 12.07)))
  (g 18.3)
  (f p)
  (f q)
  (g r t))

(let ((p 52.8) (q 35.9) (r (f 12.07))) (g 18.3) (f p) (f q) (g r t))
```

In the first case, indentation makes clear which are the bindings and which are the forms. Even if the reader doesn't know about the different roles played by the two parts of the `LET` form, the indentation suggests a difference.

In the second case, you'll have to count parentheses to know where the bindings end and the forms begin. Even worse, the absence of indentation destroys visual cues about the different roles played by the two parts of the `LET` form.

If you define a variable using `SETQ` and then name the same variable in a `LET` form, the value defined by `LET` supersedes the other value during evaluation of the `LET`:

```
⊙ (setq a 89)
→ 89

⊙ a
→ 89

⊙ (let ((a 3))
    (+ a 2))
→ 5

⊙ a
→ 89
```

Unlike `SETQ`, which assigns values in left-to-right order, `LET` binds variables all at the same time:

```
⊙ (setq w 77)
→ 77

⊙ (let ((w 8)
        (x w))
    (+ w x))
→ 85
```

`LET` bound `W` to `8` and `X` to `W`. Because these bindings happened at the same time, `W` still had its value of `77`.

> Lisp has a variation of `LET`, called `LET*`, that *does* perform bindings in order:
>
> ```
> ⊙ (setq u 37)
> → 37
>
> ⊙ (let* ((v 4)
>          (u v))
>     (+ u v))
> ```

```
    → 8
```

COND

The `COND` macro lets you evaluate Lisp forms conditionally. Like `LET`, `COND` uses parentheses to delimit different parts of the form. Consider these examples:

```
⊙ (let ((a 1)
        (b 2)
        (c 1)
        (d 1))
    (cond ((eql a b) 1)
          ((eql a c) "First form" 2)
          ((eql a d) 3)))
→ 2
```

In the above `COND` form we defined three clauses. Each clause is a list beginning with a test form and followed by as many body forms as desired. The body forms are simply code that you want to execute if the test succeeds. The clauses are selected in order -- as soon as one test succeeds, the corresponding body forms are evaluated and the value of the last body form becomes the value of the `COND` form.

> `COND` is more general than the special form, `IF`, which only allows one test and one form each for the *then* and *else* parts.

Let's look at what happened in the example. `EQL` returns `T` if its two arguments are identical, or the same number (there's a subtle difference that we'll cover in Chapter 17). Only two of the three tests executed. The first, `(EQL A B)`, returned NIL. Therefore, the rest of that clause (containing the number `1` as its only form) was skipped. The second clause tested `(EQL A C)`, which was true. Because this test returned a non-`NIL` value, the remainder of the clause (the two atomic forms, `"First form"` and `2`) was evaluated, and the value of the last form was returned as the value of the `COND`, which was then returned as the value of the enclosing `LET`. The third clause was never tested, since an earlier clause had already been chosen -- clauses are tested in order.

Conventional use of `COND` uses `T` as the test form in the final clause. This guarantees that the body forms of the final clause get evaluated if the tests fail in all of the other clauses. You can use the last clause to return a default value or perform some appropriate operation. Here's an example:

```
⊙ (let ((a 32))
    (cond ((eql a 13) "An unlucky number")
          ((eql a 99) "A lucky number")
          (t "Nothing special about this number")))
→ "Nothing special about this number"
```

QUOTE

Sometimes we'd like to suppress Lisp's normal evaluation rules. One such case is when we'd like a symbol to stand for itself, rather than its value, when it appears as an argument of a function call:

```
⊙ (setq a 97)
→ 97

⊙ a
→ 97
```

```
⊙ (setq b 23)
→ 23

⊙ (setq a b)
→ 23

⊙ a
→ 23

⊙ (setq a (quote b))
→ B

⊙ a
→ B
```

The difference is that `B`'s *value* is used in `(SETQ A B)`, whereas `B` *stands for itself* in `(SETQ A (QUOTE B))`.

The `QUOTE` form is so commonly used that Lisp provides a shorthand notation:

`(QUOTE` *form*`)` ≡ `'`*form*

The ≡ symbol means that the two Lisp forms are equivalent. Lisp arranges the equivalence of `'` and `QUOTE` through a reader macro. We'll take a brief look at how you can define your own reader macros in Lesson 12.

Contents | Cover
Chapter 2 | Chapter 3, Introduction | Chapter 3, Lesson 2 | Chapter 3, Lesson 3 | Chapter 3, Lesson 4 | Chapter 4

Copyright © 1995-2001, David B. Lamkins
All Rights Reserved Worldwide

This book may not be reproduced without the written consent of its author. Online distribution is restricted to the author's site.

Chapter 3 - Essential Lisp in Twelve Lessons

Lesson 4 - Putting things together, and taking them apart

CONS

`CONS` is the most basic *cons*tructor of lists. It is a function, so it evaluates both of its arguments. The second argument must be a list or `NIL`.

```
⊝ (cons 1 nil)
→ (1)

⊝ (cons 2 (cons 1 nil))
→ (2 1)

⊝ (cons 3 (cons 2 (cons 1 nil)))
→ (3 2 1)
```

`CONS` adds a new item to the beginning of a list. The empty list is equivalent to `NIL`,

```
() ≡ NIL
```

so we could also have written:

```
⊝ (cons 1 ())
→ (1)

⊝ (cons 2 (cons 1 ()))
→ (2 1)

⊝ (cons 3 (cons 2 (cons 1 ())))
→ (3 2 1)
```

> In case you're wondering, yes, there's something special about `NIL`. `NIL` is one of two symbols in Lisp that isn't a keyword but still has itself as its constant value. `T` is the other symbol that works like this.
>
> The fact that `NIL` evaluates to itself, combined with `() ≡ NIL`, means that you can write `()` rather than `(QUOTE ())`. Otherwise, Lisp would have to make an exception to its evaluation rule to handle the empty list.

LIST

As you may have noticed, building a list out of nested `CONS` forms can be a bit tedious. The `LIST` form does the same thing in a more perspicuous manner:

```
⊝ (list 1 2 3)
→ (1 2 3)
```

`LIST` can take any number of arguments. Because `LIST` is a function, it evaluates its arguments:

```
⊝ (list 1 2 :hello "there" 3)
→ (1 2 :HELLO "there" 3)
```

```
⊙ (let ((a :this)
        (b :and)
        (c :that))
     (list a 1 b c 2))
→ (:THIS 1 :AND :THAT 2)
```

FIRST and REST

If you think of a list as being made up of two parts -- the first element and everything else -- then you can retrieve any individual element of a list using the two operations, `FIRST` and `REST`.

```
⊙ (setq my-list (quote (1 2 3 4 5)))
→ (1 2 3 4 5)

⊙ (first my-list)
→ 1

⊙ (rest my-list)
→ (2 3 4 5)

⊙ (first (rest my-list))
→ 2

⊙ (rest (rest my-list))
→ (3 4 5)

⊙ (first (rest (rest my-list)))
→ 3

⊙ (rest (rest (rest my-list)))
→ (4 5)

⊙ (first (rest (rest (rest my-list))))
→ 4
```

Clearly, chaining together `FIRST` and `REST` functions could become tedious. Also, the approach can't work when you need to select a particular element when the program runs, or when the list is of indeterminate length. We'll look at how to solve these problems in Chapter 4 by defining recursive functions. Later, in Chapter 13, we'll see the functions that Lisp provides to perform selection on the elements of lists and other sequences.

> `FIRST` and `REST` are fairly recent additions to Lisp, renaming the equivalent functions `CAR` and `CDR`, respectively. `CAR` and `CDR` got their names from an implementation detail of one of the earliest Lisp implementations, and the names persisted for decades despite the fact that the underlying implementation had long since changed.

Contents | Cover
Chapter 2 | Chapter 3, Introduction |Chapter 3, Lesson 3 | Chapter 3, Lesson 4 | Chapter 3, Lesson 5 | Chapter 4 |

Copyright © 1995-2001, David B. Lamkins
All Rights Reserved Worldwide

This book may not be reproduced without the written consent of its author. Online distribution is restricted to

the author's site.

Chapter 3 - Essential Lisp in Twelve Lessons

Lesson 5 - Naming and Identity

A symbol is just a name

A symbol is just a name. It can stand for itself. This makes it easy to write certain kinds of programs in Lisp. For example, if you want your program to represent relationships in your family tree, you can make a database that keeps relationships like this:

```
(father John Barry)
(son John Harold)
(father John Susan)
(mother Edith Barry)
(mother Edith Susan)
...
```

Each relationship is a list. `(father John Barry)` means that John is Barry's father. Every element of every list in our database is a symbol. Your Lisp program can compare symbols in this database to determine, for example, that Harold is Barry's grandfather. If you tried to write a program like this in another language -- a language without symbols -- you'd have to decide how to represent the names of family members and relationships, and then create code to perform all the needed operations -- reading, printing, comparison, assignment, etc. This is all built into Lisp, because symbols are a data type distinct from the objects they might be used to name.

A symbol is always unique

Every time your program uses a symbol, that symbol is *identical* to every other symbol with the same name. You can use the `EQ` test to compare symbols:

```
⊙ (eq 'a 'a)
→ T

⊙ (eq 'david 'a)
→ NIL

⊙ (eq 'David 'DAVID)
→ T

⊙ (setq zzz 'sleeper)
→ SLEEPER

⊙ (eq zzz 'sleeper)
→ T
```

Notice that it doesn't matter whether you use uppercase or lowercase letters in your symbol names. Internally, Lisp translates every alphabetic character in a symbol name to a common case -- usually upper, but you can control this by setting a flag in the Lisp reader.

> When you learn about packages in Lesson 10 (also see Chapter 31), you can create symbol names that are not identical given the same spelling. For now, all you need to know is that any symbol spelled with a `:` gets special treatment.

A symbol can name a value

Although the ability for a Lisp symbol to stand for itself is sometimes useful, a more common use is for the symbol to name a value. This is the role played by variable and function names in other programming languages. A Lisp symbol most commonly names a value or -- when used as the first element of a function call form -- a function.

What's unusual about Lisp is that a symbol can have a value as a function and a variable at the same time:

```
⊙ (setq first 'number-one)
→ NUMBER-ONE

⊙ (first (list 3 2 1))
→ 3

⊙ first
→ NUMBER-ONE
```

Note how `FIRST` is used as a variable in the first and last case, and as a function (predefined by Lisp, in this example) in the second case. Lisp decides which of these values to use based on where the symbol appears. When the evaluation rule requires a value, Lisp looks for the variable value of the symbol. When a function is called for, Lisp looks for the symbol's function.

A symbol can have other values besides those it has as a variable or function. A symbol can also have values for its documentation, property list, and print name. A symbol's documentation is text that you create to describe a symbol. You can create this using the `DOCUMENTATION` form or as part of certain forms which define a symbol's value. Because a symbol can have multiple meanings, you can assign documentation to each of several meanings, for example as a function and as a variable.

A property list is like a small database with a single key per entry. We'll look at this use of symbols in Lesson 10.

The print name is what Lisp uses to print the symbol. You normally don't want to change this; if you do, Lisp will print the symbol with a different name than it originally used to read the symbol, which will create a different symbol when later read by Lisp.

A value can have more than one name

A value can have more than one name. That is, more than one symbol can share a value. Other languages have pointers that work this way. Lisp does not expose pointers to the programmer, but does have shared objects. An object is considered identical when it passes the `EQ` test. Consider the following:

```
⊙ (setq L1 (list 'a 'b 'c))
→ (A B C)

⊙ (setq L2 L1)
→ (A B C)

⊙ (eq L1 L2)
→ T

⊙ (setq L3 (list 'a 'b 'c))
→ (A B C)
```

```
⊙ (eq L3 L1)
→ NIL
```

Here, `L1` is `EQ` to `L2` because `L1` names the *same value* as `L2`. In other words, the value created by the `(LIST 'A 'B 'C)` form has two names, `L1` and `L2`. The `(SETQ L2 L1)` form says, "Make the value of `L2` be the value of `L1`." Not a copy of the the value, but the value. So `L1` and `L2` *share* the same value -- the list `(A B C)` which was first assigned as the value of `L1`.

`L3` also has a list `(A B C)` as its value, but it is a *different* list than the one shared by `L1` and `L2`. Even though the value of `L3` *looks* the same as the value of `L1` and `L2`, it is a different list because it was created by a different `LIST` form. So `(EQ L3 L1)`→`NIL` because their values are different lists, each made of the symbols `A`, `B`, and `C`.

Contents | Cover
Chapter 2 | Chapter 3, Introduction | Chapter 3, Lesson 4 | Chapter 3, Lesson 5 | Chapter 3, Lesson 6 | Chapter 4 |

Copyright © 1995-2001, David B. Lamkins
All Rights Reserved Worldwide

This book may not be reproduced without the written consent of its author. Online distribution is restricted to the author's site.

Chapter 3 - Essential Lisp in Twelve Lessons

Lesson 6 - Binding versus Assignment

Binding creates a new place to hold a value

Lisp often "creates a binding" for a variable by allocating a piece of storage to hold the variable's value and putting the value into the newly allocated memory. Binding is a very general mechanism for implementing lexical scope for variables, but it has other uses depending upon the lifetime of the binding. We'll revisit this in Chapter 8 when we study lifetime and visibility.

> Yes, Lisp allocates storage for new bindings. While this sounds like it could be horribly inefficient, we've said nothing yet about *where* Lisp allocated the storage. For example, Lisp binds function parameters to actual values, but allocates the storage on the stack just like any other programming language. Lisp creates bindings in the heap if it can't determine that the binding has a lifetime which ends when the binding form finishes executing.

Bindings have names

Lisp gives each binding a name. Otherwise, how would your program refer to the binding? Simple, eh? Hold on...

A binding can have different values at the same time

It is quite common for multiple bindings to share the same name. For example:

```
(let ((a 1))
   (let ((a 2))
      (let ((a 3))
         ...)))
```

Here, `A` has three distinct bindings by the time the body (marked by `...`) executes in the innermost `LET`.

> This is not to say that the above example is representative of typical Lisp code, however.

One binding is innermost

```
;; Here, A has no binding.
(let ((a 1))
   ;; Here, the innermost binding of A has the value 1.
   (let ((a 2))
      ;; Here, the innermost binding of A has the value 2.
      (let ((a 3))
         ;; Here, the innermost binding of A has the value 3.
         ...)))
```

As you can see, the notion of innermost binding depends on the relative position of your program's code to the form that established a particular binding. If you look at how binding forms are nested (easy to do if you indent your code as shown above) then the program has access to bindings created around, or enclosing, your program code.

One more thing you should know is that an outer binding is still visible through inner binding forms, as long as the inner binding form does not bind the same symbol:

```
;; Here, A and B have no binding.
(let ((a 1)
      (b 9))
   ;; Here, the innermost binding of A has the value 1,
   ;; and the binding of B has the value 9.
   (let ((a 2))
      ;; Here, the innermost binding of A has the value 2.
      ;; The binding of B still has the value 9.
      (let ((a 3))
         ;; Here, the innermost binding of A has the value 3.
         ;; B still has the value 9 from the outermost LET form.
         ...)))
```

The program can only access bindings it creates

When a binding form binds a new value to an existing symbol, the previous value becomes shadowed. The value of the outer binding is hidden (but not forgotten) while your program code executes inside the inner binding form. But as soon as your program leaves the inner binding form, the value of the outer binding is restored. For example:

```
(let ((z 1))
   ;; Here, the innermost binding of Z has the value 1.
   (let ((z 2))
      ;; Here, the innermost binding of Z has the value 2.
      ...)
   ;; Now we're outside the inner binding form,
   ;; and we again see the binding with the value 1.
   ...)
```

Assignment gives an old place a new value

The `SETQ` form changes the value of an existing binding:

```
(let ((z 1))
   ;; Here, the innermost binding of Z has the value 1.
   (setq z 9)
   ;; Now the value of Z is 9.
   (let ((z 2))
      ;; Here, the innermost binding of Z has the value 2.
      ...)
   ;; Now we're outside the inner binding form,
   ;; and we again see the outer binding of Z with the value 9.
   ...)
```

The `SETQ` form above changed the value of the outer binding of `Z` for the remainder of the outer `LET` form. This is often the wrong thing to do. The problem is that you now have to look in two places to discover the value of `Z` -- first at the binding forms, then in the program code for assignments such as `SETQ`. While the binding forms are indented by convention (many Lisp editors do this as you type), the assignment form, as part of the body code of the program, gets no special indentation; this makes it harder to spot when you read the program.

We can quite easily avoid the assignment in the previous example by introducing a new binding:

```
(let ((z 1))
   ;; Here, the innermost binding of Z has the value 1.
   (let ((z 9))
      ;; Now the value of Z is 9.
      (let ((z 2))
         ;; Here, the innermost binding of Z has the value 2.
         ...)
      ;; Now we're outside the innermost binding form,
      ;; and we again see the middle binding of Z with the value 9.
      ...)
   ;; Here, we see the outermost binding of Z with the value 1.
   ...)
```

Now all of the bindings of `Z` are apparent from the relative indentation of the `LET` forms. While reading the program, all we have to do to find the right binding for `Z` at any point in our program code (the `...` in the example) is to scan vertically looking for a `LET` form at an outer level of indentation.

When a `SETQ` form refers to a variable that is not bound by an enclosing `LET` form, it assigns a value to the global or special value of the symbol. A global value is accessible anywhere it's not shadowed, and stays available for as long as the Lisp system runs. We'll look at special variables in Chapter 8.

```
(setq a 987)
;; Here, A has the global value 987.
(let ((a 1))
   ;; Here, the binding of A to the value 1 shadows the global value.
   ...)
;; Now the global value of A is again visible.
...
```

Contents | Cover
Chapter 2 | Chapter 3, Introduction | Chapter 3, Lesson 5 | Chapter 3, Lesson 6 | Chapter 3, Lesson 7 | Chapter 4 |

Copyright © 1995-2001, David B. Lamkins
All Rights Reserved Worldwide

This book may not be reproduced without the written consent of its author. Online distribution is restricted to the author's site.

Chapter 3 - Essential Lisp in Twelve Lessons

Lesson 7 - Essential Function Definition

DEFUN defines named functions

You can define a named function using the `DEFUN` form:

```
⊝ (defun secret-number (the-number)
     (let ((the-secret 37))
        (cond ((= the-number the-secret) 'that-is-the-secret-number)
              ((< the-number the-secret) 'too-low)
              ((> the-number the-secret) 'too-high))))
→ SECRET-NUMBER
```

> We described `LET`, `COND`, and `'` (a.k.a. `QUOTE)` in Lesson 3. The numeric comparison functions have the obvious meaning.

The `DEFUN` form has three arguments:

1. the name of the function: `SECRET-NUMBER`,
2. a list of argument names: `(THE-NUMBER)`, which will be bound to the function's parameters when it is called, and
3. the body of the function: `(LET ...)`.

Since all three of these should stand for themselves, `DEFUN` does not evaluate any of its arguments. (If it did, you'd face the inconvenience of having to quote each argument.)

`DEFUN` returns the name of the defined function, and installs a global definition using the name, parameter list, and body that you supplied. Once you create a function using `DEFUN`, you can use it right away:

```
⊝ (secret-number 11)
→ TOO-LOW

⊝ (secret-number 99)
→ TOO-HIGH

⊝ (secret-number 37)
→ THAT-IS-THE-SECRET-NUMBER
```

When you call the function, its parameter (e.g. `99` in the second example) is bound to the argument name (i.e. `THE-NUMBER`) you supplied in the definition. Then, the body of the function (i.e. `(LET ...)`) is evaluated within the context of the parameter binding. In other words, evaluating `(SECRET-NUMBER 99)` causes the body of the `SECRET-NUMBER` function definition to be executed with the variable `THE-NUMBER` bound to `99`.

Of course, you can define a function of more than one argument:

```
⊝ (defun my-calculation (a b c x)
     (+ (* a (* x x)) (* b x) c))
→ MY-CALCULATION

⊝ (my-calculation 3 2 7 5)
```

```
→ 92
```

When calling a function, parameters are bound to argument names in order. Lisp has several optional variations on the list of argument names. Formally, this list is called a *lambda list* -- we'll examine some of its other features in Chapter 21.

LAMBDA defines anonymous functions

At times you'll need a function in only one place in your program. You could create a function with `DEFUN` and call it just once. Sometimes, this is the best thing to do, because you can give the function a descriptive name that will help you read the program at some later date. But sometimes the function you need is so trivial or so obvious that you don't want to have to invent a name or worry about whether the name might be in use somewhere else. For situations like this, Lisp lets you create an unnamed, or anonymous, function using the `LAMBDA` form. A `LAMBDA` form looks like a `DEFUN` form without the name:

```
(lambda (a b c x)
   (+ (* a (* x x)) (* b x) c))
```

You can't evaluate a `LAMBDA` form; it must appear only where Lisp expects to find a function -- normally as the first element of a form:

```
⊙ (lambda (a b c x)
     (+ (* a (* x x)) (* b x) c))
→| Error

⊙ ((lambda (a b c x)
      (+ (* a (* x x)) (* b x) c))
   3 2 7 5)
→ 92
```

Contents | Cover
Chapter 2 | Chapter 3, Introduction | Chapter 3, Lesson 6 | Chapter 3, Lesson 7 | Chapter 3, Lesson 8 | Chapter 4 |

Copyright © 1995-2001, David B. Lamkins
All Rights Reserved Worldwide

This book may not be reproduced without the written consent of its author. Online distribution is restricted to the author's site.

Chapter 3 - Essential Lisp in Twelve Lessons

Lesson 8 - Essential Macro Definition

DEFMACRO defines named macros

A `DEFMACRO` form looks a lot like a `DEFUN` form (see Lesson 7) -- it has a name, a list of argument names, and a body:

```
(defmacro name (argument ...)
   body)
```

Macros return a form, not values

The macro body returns a form to be evaluated. In other words, you need to write the body of the macro such that it returns a form, not a value. When Lisp evaluates a call to your macro, it first evaluates the body of your macro definition, then evaluates the result of the first evaluation. (By way of comparison, a function's body is evaluated to return a value.)

Here are a couple of simple macros to illustrate most of what you need to know:

```
⊙ (defmacro setq-literal (place literal)
     `(setq ,place ',literal))
→ SETQ-LITERAL

⊙ (setq-literal a b)
→ B

⊙ a
→ B

⊙ (defmacro reverse-cons (rest first)
     `(cons ,first ,rest))
→ REVERSE-CONS

⊙ (reverse-cons nil A)
→ (B)
```

`SETQ-LITERAL` works like `SETQ`, except that *neither* argument is evaluated. (Remember that `SETQ` evaluates its second argument.) The body of `SETQ-LITERAL` has a form that begins with a ` (pronounced "backquote"). Backquote behaves like quote -- suppressing evaluation of all the enclosed forms -- except where a comma appears within the backquoted form. A symbol following the comma is evaluated.

So in our call to `(SETQ-LITERAL A B)` above, here's what happens:

1. bind `PLACE` to the symbol `A`.
2. bind `LITERAL` to the symbol `B`.
3. evaluate the body `` `(SETQ ,PLACE ',LITERAL) ``, following these steps:
    1. evaluate `PLACE` to get the symbol `A`.
    2. evaluate `LITERAL` to get the symbol `B`.
    3. return the form `(SETQ A 'B)`.
4. evaluate the form `(SETQ A 'B)`.

Neither the backquote nor the commas appear in the returned form. Neither `A` nor `B` is evaluated in a call to `SETQ-LITERAL`, but for different reasons. `A` is unevaluated because it appears as the first argument of `SETQ`. `B` is unevaluated because it appears after a quote in the form returned by the macro.

The operation of `(REVERSE-CONS NIL A)` is similar:

1. bind REST

   to the symbol `NIL`.
2. bind `FIRST` to the symbol `A`.
3. evaluate the body `` `(CONS ,FIRST ,REST) ``, following these steps:
    1. evaluate `FIRST` to get the symbol `A`.
    2. evaluate `REST` to get the symbol `NIL`.
    3. return the form `(CONS A NIL).`
4. evaluate the form `(CONS A NIL).`

Both arguments of `REVERSE-CONS` are evaluated because `CONS` evaluates its arguments, and our macro body doesn't quote either argument. `A` evaluates to the symbol `B`, and `NIL` evaluates to itself.

If you want to see how your macro body appears before evaluation, you can use the `MACROEXPAND` function:

```
⊙ (macroexpand '(setq-literal a b))
→ (SETQ A 'B)

⊙ (macroexpand '(reverse-cons nil a))
→ (CONS A NIL)
```

Since `MACROEXPAND` is a function, it evaluates its arguments. This is why you have to quote the form you want expanded.

The examples in this lesson are deliberately very simple, so you can understand the basic mechanism. In general, macros are trickier to write than functions -- in Chapter 20 we'll look at the reasons and the correct techniques for dealing with more complex situations.

Contents | Cover
Chapter 2 | Chapter 3, Introduction | Chapter 3, Lesson 7 | Chapter 3, Lesson 8 | Chapter 3, Lesson 9 | Chapter 4 |

Copyright © 1995-2001, David B. Lamkins
All Rights Reserved Worldwide

This book may not be reproduced without the written consent of its author. Online distribution is restricted to the author's site.

Chapter 3 - Essential Lisp in Twelve Lessons

Lesson 9 - Essential Multiple Values

Most forms create only one value

A form typically returns only one value. Lisp has only a small number of forms which create or receive multiple values.

VALUES creates multiple (or no) values

The `VALUES` form creates zero or more values:

```
⊙ (values)

⊙ (values :this)
→ :THIS

⊙ (values :this :that)
→ :THIS
→ :THAT
```

We show how many values are returned by the number of → lines produced by the evaluation of the form. The three `VALUES` forms in the example above produced zero, one, and two values, respectively.

`VALUES` is a function, and so evaluates its arguments.

A few special forms receive multiple values

What might you want to do with multiple values in a program? The most basic operations are to:

1. bind each value to a separate symbol, or
2. collect the values into a list.

Use `MULTIPLE-VALUE-BIND` to bind each value to a separate symbol:

```
⊙ (multiple-value-bind (a b c) (values 2 3 5)
     (+ a b c))
→ 10
```

If you provide more values than symbols, the excess values are ignored:

```
⊙ (multiple-value-bind (a b c) (values 2 3 5 'x 'y)
     (+ a b c))
→ 10
```

If you provide fewer values than symbols, the excess symbols are bound to `NIL`:

```
⊙ (multiple-value-bind (w x y z) (values :left :right)
     (list w x y z))
→ (:LEFT :RIGHT NIL NIL)
```

Some forms pass along multiple values

Some forms pass along the last value in their body, rather than creating a new value. Examples include the bodies of `LET`, `COND`, `DEFUN`, and `LAMBDA`.

```
⊙ (let ((a 1)
        (b 2))
     (values a b))
→ 1
→ 2

⊙ (cond (nil 97)
        (t (values 3 4)))
→ 3
→ 4

⊙ (defun foo (p q)
     (values (list :p p) (list :q q)))
→ FOO

⊙ (foo 5 6)
→ (:P 5)
→ (:Q 6)

⊙ ((lambda (r s)
      (values r s))
     7 8)
→ 7
→ 8
```

> In the case of the function and lambda bodies, the multiple values are actually returned by something called an "*implicit* `PROGN`." This is a fancy way of saying that the bodies can contain multiple forms, and only the value of the last form is returned.
>
> You can use the `PROGN` special form when you want this behavior. `(PROGN form1 form2 ... formN)` evaluates `form1` through `formN` in order, and returns the value of `formN`.

Contents | Cover
Chapter 2 | Chapter 3, Introduction | Chapter 3, Lesson 8 | Chapter 3, Lesson 9 | Chapter 3, Lesson 10 | Chapter 4 |

Copyright © 1995-2001, David B. Lamkins
All Rights Reserved Worldwide

This book may not be reproduced without the written consent of its author. Online distribution is restricted to the author's site.

Chapter 3 - Essential Lisp in Twelve Lessons

Lesson 10 - A Preview of Other Data Types

Lisp almost always does the right thing with numbers

This sounds like a strange thing to say. Don't computers always do the right thing with numbers? Well, no... Not usually.

Numeric calculations can break in lots of different ways. One of the biggest trouble spots is in calculations with floating point numbers (your language may call them *real* numbers, but that's a lie). There are probably half as many books written on proper use of floating point calculations as there are on visual- or object-oriented-anything -- and that's a lot.

The problem with floating point numbers is that they're not mathematically accurate real numbers, but are often (mis)used as if they are. The main problem is that floating point numbers have a limited accuracy -- only so many digits to the right of the decimal point. Now, if all of the numbers in a calculation are of approximately the same magnitude, then the calculation won't lose accuracy. But if the numbers are of very different magnitude, then a floating point calculation sacrifices accuracy.

> Suppose that a floating point number on your computer can accurately represent 7 decimal digits. Then you can add 1897482.0 to 2973225.0 and get a completely accurate answer. But if you try to add 1897482.0 to 0.2973225, the accurate answer has fourteen digits, while your computer will answer with 1897482.0.

The other problem with floating point numbers is more subtle. When you write a program, you write numbers in base 10. But the computer does all arithmetic in base 2. The conversion from base 10 to base 2 does funny things to certain "obviously exact" numbers. For example, the decimal number 0.1 is a repeating fraction when translated into binary. Because the computer can't store the infinite number of digits required by a repeating fraction, it can't store the number 0.1 accurately.

Integer (whole number) arithmetic poses another problem in most computer languages -- they tend to impose a limit on the maximum positive or negative value that an integer can hold. So, if you try to add the number one to the largest integer your language lets the computer handle, one of two things will happen:

1. your program will terminate with an error, or
2. you'll get a wildly incorrect answer (the largest positive number plus one yields the largest negative integer in at least one computer language).

So how does Lisp manage to do the right thing with numbers? After all, it seems like these problems are inherent in computer arithmetic. The answer is that Lisp doesn't do use just the built-in computer arithmetic operations -- it adds certain mathematically accurate numeric data types:

- *bignums* are integers with an unlimited number of digits (subject only to limitations of computer memory)
- *rational numbers* are the exact quotient of two integers, not a floating point number resulting from an approximate machine division algorithm

Of course, Lisp also has machine-based integers and floating point numbers. Machine integers are called *fixnums* in Lisp. So long as a whole number falls within the numeric range of a fixnum, Lisp will store it as a machine integer. But if it gets too big, Lisp automatically promotes it to a bignum.

When I said that Lisp almost always does the right thing with numbers, I meant that it *almost always* chooses the numeric representation that is mathematically correct:

```
⊙ (/ 1 3)
→ 1/3

⊙ (+ (/ 7 11) (/ 13 31))
→ 360/341

⊙ (defun factorial (n)
    (cond ((= n 0) 1)
          (t (* n (factorial (- n 1))))))
→ FACTORIAL

⊙ (factorial 100)
→ 933262154439441526816992388562667004907159682643816214685
   929638952175999932299156089414639761565182862536979208272
   23758251185210916864000000000000000000000000
```

You can write calculations to use floating point numbers, but Lisp won't automatically turn an exact numeric result into an inexact floating point number -- you have to ask for it. Floating point numbers are *contagious* -- once you introduce one into a calculation, the result of the entire calculation stays a floating point number:

```
⊙ (float (/ 1 3))
→ 0.3333333333333333

⊙ (* (float (/ 1 10)) (float (/ 1 10)))
→ 0.010000000000000002

⊙ (+ 1/100 (* (float (/ 1 10)) (float (/ 1 10))))
→ 0.020000000000000004

⊙ (+ 1/100 1/100) ; compare to previous calculation
→ 1/50

⊙ (* 3 7 10.0)
→ 210.0

⊙ (- 1.0 1)
→ 0.0

⊙ (+ 1/3 2/3 0.0)
→ 1.0

⊙ (+ 1/3 2/3)
→ 1 ; compare to previous calculation
```

Lisp prints floating point numbers with a decimal point, and integers without.

Characters give Lisp something to read and write

Basic Lisp I/O uses characters. The `READ` and `WRITE` functions turn characters into Lisp objects and vice versa. `READ-CHAR` and `WRITE-CHAR` read and write single characters.

```
⊙ (read)
▪ a↵
→ A

⊙ (read)
▪ #\a↵
→ a

⊙ (read-char)
▪ a
→ #\a

⊙ (write 'a)
⇒ A
→ A

⊙ (write #\a)
⇒ #\a
→ #\a

⊙ (write-char #\a)
⇒ a
→ #\a

⊙ (write-char 'a)
⇥ Error: Not a character
```

We've introduced some new notation in the above examples. The ▪ symbol means that Lisp expects input in response to an input function such as `READ`. This is different from ⊙, which accepts input to be evaluated and printed. The ↵ symbol indicates a newline character, generated by the **`return`** or **`enter`** key.

The ⇒ indicates output that is printed rather than returned as a value.

You should notice that newline terminates `READ` input. This is because `READ` collects characters trying to form a complete Lisp expression. We'll see more of this in Lesson 11. In the example, `READ` collects a symbol, which is terminated by the newline. The symbol could also have been terminated by a space, a parenthesis, or any other character that can't be part of a symbol.

In contrast, `READ-CHAR` reads exactly one character from the input. As soon as that character is consumed, `READ-CHAR` completes executing and returns the character.

> Some Lisp systems systems may require you to press the **`return`** key before any input is recognized. This is unusual, and can often be fixed by a configuration parameter -- consult your Lisp vendor.

`WRITE` and `WRITE-CHAR` both return the value they're given. The way in which they print the value is different. `WRITE` prints the value so that it could be presented to `READ` to create the same value. `WRITE-CHAR` prints just the readable character, without the extra Lisp syntax (the `#\`) that would identify it to `READ` as a character.

Lisp represents a single character using the notation `#\`*`char`*, where *char* is a literal character or the name of a character that does not have a printable glyph.

Character	Hex Value	Lisp	Standard?
space	20	#\Space	yes

newline	--	#\Newline	yes
backspace	08	#\Backspace	semi
tab	09	#\Tab	semi
linefeed	0A	#\Linefeed	semi
formfeed	0C	#\Page	semi
carriage return	0D	#\Return	semi
rubout or DEL	7F	#\Rubout	semi

Only `#\Space` and `#\Newline` are required on all Lisp systems. Systems that use the ASCII character set will probably implement the rest of the character codes shown above.

The `#\Newline` character stands for whatever convention represents the end of a printed line on the host system, e.g.:

System	Newline	Hex Value
Macintosh	CR	0D
MS-DOS	CR LF	0D 0A
Unix	LF	0A

The 94 printable standard characters are represented by #*char*:

```
  ! " # $ % & ' ( ) * + , - . /
0 1 2 3 4 5 6 7 8 9 : ; < = > ?
@ A B C D E F G H I J K L M N O
P Q R S T U V W X Y Z [ \ ] ^ _
` a b c d e f g h i j k l m n o
p q r s t u v w x y z { | } ~
```

Arrays organize data into tables

If you need to organize data in tables of two, three, or more dimensions, you can create an array:

```
⊙ (setq a1 (make-array '(3 4)))
→ #2A((NIL NIL NIL NIL)
      (NIL NIL NIL NIL)
      (NIL NIL NIL NIL))

⊙ (setf (aref a1 0 0) (list 'element 0 0))
→ (ELEMENT 0 0)

⊙ (setf (aref a1 1 0) (list 'element 1 0))
→ (ELEMENT 1 0)

⊙ (setf (aref a1 2 0) (list 'element 2 0))
→ (ELEMENT 2 0)

⊙ a1
→ #2A(((ELEMENT 0 0) NIL NIL NIL)
      ((ELEMENT 1 0) NIL NIL NIL)
      ((ELEMENT 2 0) NIL NIL NIL))

⊙ (aref a1 0 0)
→ (ELEMENT 0 0)

⊙ (setf (aref a1 0 1) pi)
→ 3.141592653589793
```

```
⊙ (setf (aref a1 0 2) "hello")
→ "hello"

⊙ (aref a1 0 2)
→ "hello"
```

You create an array using `MAKE-ARRAY`, which takes a list of dimensions and returns an array. By default, an array can contain any kind of data; optional arguments let you restrict the element data types for the sake of efficiency.

An array's *rank* is the same as its number of dimensions. We created a rank-2 array in the above example. Lisp prints an array using the notation `#`*`rank`*`A(...)`. The contents of the array appear as nested lists, with the first dimension appearing as the outermost grouping, and the last dimension appearing as the elements of the innermost grouping.

> Your Lisp system will probably not print an array with line breaks as I've shown here. I added these breaks to emphasize the structure of the array.

To retrieve an element of an array, use `AREF`. `AREF`'s first argument is the array; the remaining arguments specify the index along each dimension. The number of indices must match the rank of the array.

To set an element of an array, use `AREF` inside a `SETF` form as shown in the example. `SETF` is similar to `SETQ`, except where `SETQ` assigns a value to a *symbol*, `SETF` assigns a value to a *place*. In the examples, the `AREF` form specifies the place as an element in the array.

Vectors are one-dimensional arrays

Vectors are one-dimensional arrays. You can create a vector using `MAKE-ARRAY`, and access its elements using `AREF`.

```
⊙ (setq v1 (make-array '(3)))
→ #(NIL NIL NIL)

⊙ (make-array 3)
→ #(NIL NIL NIL)

⊙ (setf (aref v1 0) :zero)
→ :ZERO

⊙ (setf (aref v1 1) :one)
→ :ONE

⊙ (aref v1 0)
→ :ZERO

⊙ v1
→ #(:ZERO :ONE NIL)
```

Lisp prints vectors using the slightly abbreviated form `#(...)`, rather than `#1A(...)`.

You can use either a single-element list or a number to specify the vector dimensions to `MAKE-ARRAY` -- the effect is the same.

You can create a vector from a list of values, using the `VECTOR` form:

```
⊙ (vector 34 22 30)
→ #(34 22 30)
```

This is similar to the `LIST` form, except that the result is a vector instead of a list. There are other similarities between lists and vectors: both are *sequences*. Sequences are manipulated by the functions we'll see in Chapter 13.

You can use `AREF` to access the elements of a vector, or you can use the sequence-specific function, `ELT`:

```
⊙ (setf v2 (vector 34 22 30 99 66 77))
→ #(34 22 30 99 66 77)

⊙ (setf (elt v2 3) :radio)
→ :RADIO

⊙ v2
→ #(34 22 30 :RADIO 66 77)
```

Strings are vectors that contain only characters

You already know how to write a string using the `"..."` syntax. Since a string is a vector, you can apply the array and vector functions to access elements of a string. You can also create strings using the `MAKE-STRING` function or change characters or symbols to strings using the `STRING` function.

```
⊙ (setq s1 "hello, there.")
→ "hello, there."

⊙ (setf (elt s1 0) #\H))
→ #\H

⊙ (setf (elt s1 12) #\!)
→ #\!

⊙ s1
→ "Hello, there!"

⊙ (string 'a-symbol)
→ "A-SYMBOL"

⊙ (string #\G)
→ "G"
```

Symbols are unique, but they have many values

We saw in Lesson 5 that a symbol has a unique identity, but this bears repeating: A symbol is *identical* to any other symbol spelled the same way (including its package designation, which we'll learn more about at the end of this lesson). This means that you can have Lisp read a program or data, and every occurrence of a symbol with the same spelling is the same symbol. Since Lisp supplies the mechanism to do this, it's one less thing you have to worry about when you write a program that manipulates *symbolic* information.

We also learned in Lesson 5 that a symbol can have values as a variable and a function, and for documentation, print name, and properties. A symbol's property list is like a miniature database which associates a number of key/value pairs with the symbol. For example, if your program represented and manipulated objects, you could store information about an object on its property list:

```
⊙ (setf (get 'object-1 'color) 'red)
→ RED

⊙ (setf (get 'object-1 'size) 'large)
→ LARGE

⊙ (setf (get 'object-1 'shape) 'round)
→ ROUND

⊙ (setf (get 'object-1 'position) '(on table))
→ (ON TABLE)

⊙ (setf (get 'object-1 'weight) 15)
→ 15

⊙ (symbol-plist 'object-1)
→ (WEIGHT 15 POSITION (ON TABLE) SHAPE ROUND SIZE LARGE
     COLOR RED)

⊙ (get 'object-1 'color)
→ RED

⊙ object-1
→ Error: no value
```

Note that `OBJECT-1` doesn't have a value -- all of the useful information is in two places: the *identity* of the symbol, and the symbol's properties.

> This use of properties predates modern object programming by a few decades. It provides two of the three essential mechanisms of an object: identity and encapsulation (remember that property values could just as well be a function). The third mechanism, inheritance, was sometimes simulated by links to other "objects."

Properties are less often used in modern Lisp programs. Hash tables (see below), structures (described in the next section), and CLOS objects (see Chapter 7 and Chapter 14) provide all of the capabilities of property lists in ways that are easier to use and more efficient. Modern Lisp development systems often use properties to annotate a program by keeping track of certain information such as the file and file position of the defining form for a symbol, and the definition of a function's argument list (for use by informational tools in the programming environment).

Structures let you store related data

A Lisp structure gives you a way to create an object which stores related data in named slots.

```
⊙ (defstruct struct-1 color size shape position weight)
→ STRUCT-1

⊙ (setq object-2 (make-struct-1
                   :size 'small
                   :color 'green
                   :weight 10
                   :shape 'square))
→ #S(STRUCT-1 :COLOR GREEN :SIZE SMALL :SHAPE SQUARE
     :POSITION NIL :WEIGHT 10)

⊙ (struct-1-shape object-2)
→ SQUARE
```

```
⊙ (struct-1-position object-2)
→ NIL

⊙ (setf (struct-1-position object-2) '(under table))
→ (UNDER TABLE)

⊙ (struct-1-position object-2)
→ (UNDER-TABLE)
```

In the example, we defined a structure type named `STRUCT-1` with slots named `COLOR`, `SHAPE`, `SIZE`, `POSITION`, and `WEIGHT`. Then we created an instance of a `STRUCT-1` type, and assigned the instance to the variable `OBJECT-2`. The rest of the example shows how to access slots of a struct instance using accessor functions named for the structure type and the slot name. Lisp generates the make-*structname* and *structname-slotname* functions when you define a structure using `DEFSTRUCT`.

We'll look at `DEFSTRUCT`'s optional features in Chapter 6.

Type information is apparent at runtime

A symbol can be associated with any type of value at runtime. For cases where it matters, Lisp lets you query the type of a value.

```
⊙ (type-of 123)
→ FIXNUM

⊙ (type-of 123456789000)
→ BIGNUM

⊙ (type-of "hello, world")
→ (SIMPLE-BASE-STRING 12)

⊙ (type-of 'fubar)
→ SYMBOL

⊙ (type-of '(a b c))
→ CONS
```

`TYPE-OF` returns a symbol or a list indicating the type of its argument. This information can then be used to guide a program's behavior based upon the type of its arguments. The `TYPECASE` function combines the type inquiry with a `COND`-like dispatch.

> With the introduction of generic functions in CLOS (see Chapter 14), `TYPE-OF` is not as important as it once was.

Hash Tables provide quick data access from a lookup key

A hash table associates a value with a unique key. Unlike a property list, a hash table is well suited to a large number of key/value pairs, but suffers from excessive overhead for smaller sets of associations.

```
⊙ (setq ht1 (make-hash-table))
→ #<HASH-TABLE>

⊙ (gethash 'quux ht1)
→ NIL
→ NIL

⊙ (setf (gethash 'baz ht1) 'baz-value)
→ BAZ-VALUE

⊙ (gethash 'baz ht1)
→ BAZ-VALUE
→ T

⊙ (setf (gethash 'gronk ht1) nil)
→ NIL

⊙ (gethash 'gronk ht1)
→ NIL
→ T
```

You create a hash table using `MAKE-HASH-TABLE`, and access values using `GETHASH`. `GETHASH` returns two values. The first is the value associated with the key. The second is `T` if the key was found, and `NIL` otherwise. Notice the difference between the first and last `GETHASH` form in the examples above.

By default, a hash table is created so that its keys are compared using *EQ* -- this works for symbols, but not numbers or lists. We'll learn more about equality predicates in Chapter 17. For now, just remember that if you want to use numbers for keys, you must create a hash table using the form:

```
(make-hash-table :test #'eql)
```

If you want to use lists for keys, create your hash table with:

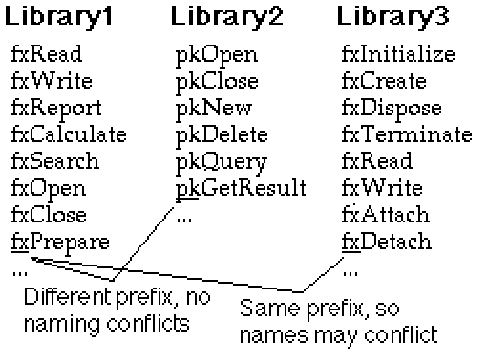
```
(make-hash-table :test #'equal)
```

If you want to remove a key, use the form `(REMHASH` *key hash-table*`)`. And if you want to change the value for a key, use `GETHASH` with `SETF`, just as if you were adding a new key/value pair.

Packages keep names from colliding

One of the things that's hard about writing programs is naming parts of your program. On one hand, you want to use names that are easy to remember and evocative of the role or purpose of the named object. On the other hand, you don't want to use a name that someone else has already used (or is likely to use) in a different program that you may someday have to make work with your program.

One way to avoid naming conflicts is to give every name in your program a unique prefix that no one else is likely to use. You see this done all the time with libraries -- the prefix is typically one to three characters. Unfortunately, this still leaves a lot of room for two software developers to choose the same prefix; especially since some prefixes are more evocative than others. If you have control over all the software that will be developed for your product, you can choose all of the prefixes and avoid problems. If you're going to buy third-party software that uses a prefix naming scheme, you'll have to work around the names chosen by your vendors and hope that two different vendors don't stumble upon the same prefix.

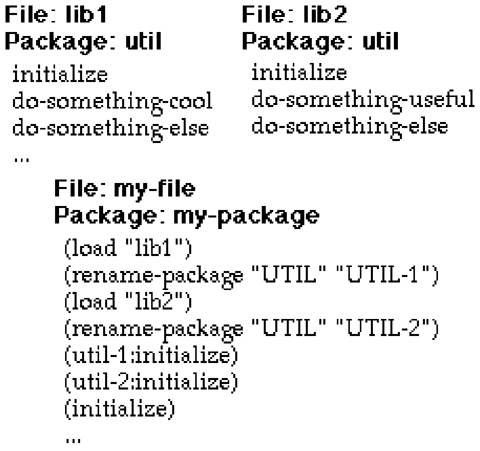


Library1	Library2	Library3
fxRead	pkOpen	fxInitialize
fxWrite	pkClose	fxCreate
fxReport	pkNew	fxDispose
fxCalculate	pkDelete	fxTerminate
fxSearch	pkQuery	fxRead
fxOpen	pkGetResult	fxWrite
fxClose	...	fxAttach
fxPrepare		fxDetach
...		...

Different prefix, no naming conflicts
Same prefix, so names may conflict

Another way to avoid naming conflicts is to use qualified names. To do this, the language must provide support for separate namespaces defined and controlled by the programmer. To understand how this works, imagine that all the names you create for your program get written on a sheet of paper with your name written at the top as a title -- this is the *qualifier* for all of your names. To see whether a name is safe to use, you only have to check the list of names you've written on this page. When someone else's software needs the services of your program, they refer to your names by using both your qualifier and name. Because the other person's software has a different qualifier, and their qualifier is implicit (that is, it doesn't need to be written) for their own names, there's no chance of a name conflict.

You might think that a qualifier is no more than a complicated way to add a prefix to a name. However, there's a subtle but important difference. A prefix is part of the name; it cannot be changed once written. A qualifier is separate from the names it qualifies, and is "written down" in exactly one place. Furthermore, you can point to the "sheet of paper" upon which names are written and refer to it as "those names." If you happen to choose the same qualifier as another programmer, you can still refer to "those names" by a qualifier of your own choosing -- In other words, you can change the qualifier *after the software has been delivered for your use*.



File: lib1 Package: util	File: lib2 Package: util
initialize	initialize
do-something-cool	do-something-useful
do-something-else	do-something-else
...	

File: my-file
Package: my-package

```
(load "lib1")
(rename-package "UTIL" "UTIL-1")
(load "lib2")
(rename-package "UTIL" "UTIL-2")
(util-1:initialize)
(util-2:initialize)
(initialize)
...
```

In the above example, two libraries are delivered in files `LIB1` and `LIB2`. Both library designers used the name `UTIL` for the name of their namespace, known in Lisp as a package name. Each library lists the names exposed to a client. The programmer who uses the two libraries writes code in the package name `MY-PACKAGE`. After loading each library, the programmer renames its package so the names are distinct. Then, names in the library are referenced using their *renamed* qualifiers, as we see in the calls to `UTIL-1:INITIALIZE` and `UTIL-2:INITIALIZE`. Notice how the `INITIALIZE` name is still available to the programmer in its unqualified form -- this is equivalent to `MY-PACKAGE:INITIALIZE`.

Lisp provides this functionality through a set of functions and macros collective known as the *package* facility. The `DEFPACKAGE` macro conveniently provides most package operations, while the `IN-PACKAGE` macro sets the current package:

```
;;;; ---- File 1 ----
(defpackage util1
  (:export init func1 func2)
  (:use common-lisp))
(in-package util1)

(defun init () 'util1-init)
(defun func1 () 'util1-func1)
(defun func2 () 'util1-func2)

;;;; ---- File 2 ----
(defpackage util2
  (:export init func1 func2)
  (:use common-lisp))
(in-package util2)

(defun init () 'util2-init)
(defun func1 () 'util2-func1)
(defun func2 () 'util2-func2)

;;;; ---- File 3 ----
(defpackage client
  (:use common-lisp)
  (:import-from util1 func1)
  (:import-from util2 func2))
(in-package client)

(defun init () 'client-init)
(util1:init)
(util2:init)
(init)
(func1)
(func2)
```

The example lists the contents of three files. File 1 and File 2 both define three functions using identical names. File 1 puts names in the `UTIL1` package, while File 2 uses the `UTIL2` package. The `DEFPACKAGE` form names the package. The `:USE` option specifies that names from another package may be used without qualification, while the `:EXPORT` option specifies the names that are exposed to clients of the package.

The `DEFPACKAGE` form only creates a package. The `USE-PACKAGE` form makes a package *current* -- all unqualified names are in whatever package is current. The `COMMON-LISP:*PACKAGE*` variable always contains the current package.

File 3 creates the `CLIENT` package. The `:IMPORT-FROM` options bring in specific names from the `UTIL1` and `UTIL2` packages -- these names may be used without qualification in the `CLIENT` package. Names that are exported from `UTIL1` or `UTIL2` but not imported by `CLIENT` may still be referenced within `CLIENT` by using an explicit qualifier of the form `package:name`.

This section covered only very basic package operations. We'll cover additional details in Chapter 31, when we look again at packages within the context of constructing large software systems.

Contents | Cover
Chapter 2 | Chapter 3, Introduction | Chapter 3, Lesson 9 | Chapter 3, Lesson 10 | Chapter 3, Lesson 11 | Chapter 4 |

Copyright © 1995-2001, David B. Lamkins
All Rights Reserved Worldwide

This book may not be reproduced without the written consent of its author. Online distribution is restricted to the author's site.

Chapter 3 - Essential Lisp in Twelve Lessons

Lesson 11 - Essential Input and Output

READ accepts Lisp data

As we saw in Lesson 10, `READ` turns characters into Lisp data. So far, you've seen a printed representation of several kinds of Lisp data:

- symbols and numbers,
- strings, characters, lists, arrays, vectors, and structures,
- and hash tables.

The Lisp reader does its job according to a classification of characters. The standard classifications are shown below. As we'll see in Lesson 12, you can alter these classifications for your own needs.

```
Standard Constituent Characters
-------------------------------
 a b c d e f g h i j k l m n o p q r s t u v w x y z
 A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
 0 1 2 3 4 5 6 7 8 9
 ! $ % & * + - . / : < = > ? @ [ ] ^ _ { } ~
 <backspace> <rubout>

Standard Terminating Macro Characters
-------------------------------------
 " ' ( ) , ; `

Standard Non-Terminating Macro Characters
-----------------------------------------
 #

Standard Single Escape Characters
---------------------------------
 \

Standard Multiple Escape Characters
-----------------------------------
 |

Standard Whitespace Characters
------------------------------
 <tab> <space> <page> <newline> <return> <linefeed>
```

If `READ` starts with a *constituent* character, it begins accumulating a symbol or number. When `READ` encounters a terminating macro character or a whitespace character, it tries to interpret the collected constituent characters first as a number, then as a symbol. If a numeric interpretation is possible, `READ` returns the number. Otherwise, `READ` changes the alphabetical characters to a standard case (normally upper case), interns the name as a symbol, and returns the symbol.

Escape characters play a special role. A single escape character forces the following character to be treated exactly as a constituent character. In this way characters that are normally treated as whitespace or terminating macro characters can be part of a symbol. If `READ` encounters an escape character, it *never* attempts to interpret the resulting constituents as a number, even if only digits were escaped.

If `READ` starts with a macro character, the character determines the next step:

"

Read a string.

'

Read a form.

(

Read a list.

;

Ignore everything up to newline.

#

Decide what to do based on the next character.

Finally, some Lisp data is not meant to be read. For example, the printed representation of a hash table looks something like `#<HASH-TABLE>`. It is an error for `READ` to attempt to read anything beginning with the characters `#<`.

PRINT writes Lisp data for you and for READ

The `PRINT` function changes a Lisp object into the sequence of characters that `READ` would need to reconstruct it:

```
⊙ (print 'abc)
⇒ ↵ABC␣
→ ABC

⊙ (print (list 1 2 3))
⇒ ↵(1 2 3)␣
→ (1 2 3)

⊙ (print "A String")
⇒ ↵"A string"␣
→ "A string"

⊙ (print 387.9532)
⇒ ↵387.9532␣
→ 387.9532

⊙ (print (make-hash-table))
⇒ ↵#<HASH-TABLE>␣
→ #<HASH-TABLE>
```

`PRINT` always begins its output with a newline character (↵), and follows its output with a space (␣). This ensures that the `PRINT` output stands apart from any surrounding output, since newline and space are both treated as whitespace, and cannot be part of the printed representation of a Lisp object (unless escaped).

Other variations of `PRINT` have different uses. `PRIN1` behaves as `PRINT`, but does not surround its output with whitespace. This might be useful if you are building up a name from successive pieces, for example. `PRINC` behaves as `PRIN1`, but generates output intended for display, rather than `READ`; for example, `PRINC` omits the quotes around a string, and does not print escape characters.

```
⊙ (print 'a\ bc)
```

```
⇒ ↵|A BC|␣
→ |A BC|

⊝ (prin1 'a\ bc)
⇒ |A BC|
→ |A BC|

⊝ (princ '|A BC|)
⇒ ↵A BC␣
→ |A BC|
```

OPEN and CLOSE let you work with files

Normally, `READ` reads from the keyboard and `PRINT` prints to the screen. Both of these functions take an optional argument; the argument specifies an input stream for `READ`, and an output stream for `PRINT`. What's a stream? A stream is a source or sink of data, typically -- but not necessarily -- characters. For now, we'll look at how text files can be the source or sink of a character stream. In Chapter 19 we'll look at some of the other possibilities.

You can attach a stream to a file using the `OPEN` function, which takes as parameters a file name and a keyword argument to specify the direction (input or output) of the stream. To finish operations on the stream and close the associated file, use the `CLOSE` function.

```
⊝ (setq out-stream (open "my-temp-file" :direction :output))
→ #<OUTPUT-STREAM "my-temp-file">

⊝ (print 'abc out-stream)
→ ABC

⊝ (close out-stream)
→ T

⊝ (setq in-stream (open "my-temp-file" :direction :input))
→ #<INPUT-STREAM "my-temp-file">

⊝ (read in-stream)
→ ABC

⊝ (close in-stream)
→ T
```

In the example, we create an output stream to the file named `my-temp-file`, and print the symbol `ABC` to that stream. Notice how `PRINT` returns its argument as usual, but doesn't print it -- the printed result has gone to the file, instead.

Next, we close the output stream and open an input stream on the same file. We then read the symbol that we printed to the file, and finish by closing the input stream.

Variations on a PRINT theme

Lisp also provides a `WRITE` function to give you control over more details of printing, using keyword arguments to control these options:

```
Keyword Argument   Default Value          Action
----------------   -------------          ------
```

`:stream`	`t`	set output stream
`:escape`	`*print-escape*`	include escape characters
`:radix`	`*print-radix*`	include radix (base) prefix
`:base`	`*print-base*`	set number base (rationals)
`:circle`	`*print-circle*`	print circular structures
`:pretty`	`*print-pretty*`	add whitespace for readability
`:level`	`*print-level*`	limit nesting depth
`:length`	`*print-length*`	limit items per nesting level
`:case`	`*print-case*`	:upper, :lower, or :mixed
`:gensym`	`*print-gensym*`	prefix uninterned symbols
`:array`	`*print-array*`	print arrays readably
`:readably`	`*print-readably*`	force printing to be readable
`:right-margin`	`*print-right-margin*`	controls pretty-printing
`:miser-width`	`*print-miser-width*`	"
`:lines`	`*print-lines*`	"
`:pprint-dispatch`	`*print-pprint-dispatch*`	"

Coincidentally, the variables named above as the default values of the keyword arguments also control the operation of `PRINT`. You can get the effect of `WRITE` with non-default keyword arguments by binding these variables in a `LET` form around a `PRIN1`:

```
(write foo                      (let ((*print-pretty* t)
   :pretty t                          (*print-right-margin* 60)
   :right-margin 60                   (*print-case* :downcase))
   :case :downcase)      ≡      (prin1 foo))
```

We used `PRIN1` rather than `PRINT` because we don't want the preceding newline and following blank that `PRINT` adds.

If your program changes the `*PRINT-...*` variables, but you need to ensure the default values at some point in your program, you can wrap that part of the program inside a `WITH-STANDARD-IO-SYNTAX` form:

```
;Define printer control for the program.
(setq *print-circle* t)
(setq *print-array* nil)
(setq *print-escape* nil)
...
;Print with the settings established above.
(print ...)
...
;Change back to default printer control settings
(with-standard-io-syntax
   ...
   ;Print with the standard settings,
   ;overriding those established above.
   (print ...)
   ...)
;Outside the WITH-STANDARD-IO-SYNTAX form,
;we once again have the print settings established
;by the SETQ forms at the top of the example.
```

Contents | Cover
Chapter 2 | Chapter 3, Introduction | Chapter 3, Lesson 10 | Chapter 3, Lesson 11 | Chapter 3, Lesson 12 | Chapter 4 |

Copyright © 1995-2001, David B. Lamkins
All Rights Reserved Worldwide

This book may not be reproduced without the written consent of its author. Online distribution is restricted to the author's site.

Chapter 3 - Essential Lisp in Twelve Lessons

Lesson 12 - Essential Reader Macros

The reader turns characters into data

We saw in Lesson 11 that the Lisp reader gathers constituent characters into symbols and numbers, and that macro characters control the reader to handle lists, strings, quoted forms, and comments. In all of these cases, the reader turns characters into data. (For reasons that will become clear shortly, a comment is just "no data.")

Standard reader macros handle built-in data types

So far, we've seen just the *standard syntax* for Lisp. This is implemented by the reader, and controlled by the *readtable*. The reader works by processing characters according to information stored in the readtable.

User programs can define reader macros

Lisp exposes the readtable through the `*readtable*` variable, and provides several functions to manipulate entries in readtables. You can use these to alter the behavior of the Lisp reader. In the following example, we change the syntax so we can write quoted (i.e. unevaluated) lists using `[` and `]`.

```
;This is wrong:
⊙ (1 2 3 4 5 6)
⇥ Error: 1 is not a function

;Should have done this, instead:
⊙ '(1 2 3 4 5 6)
→ (1 2 3 4 5 6)

;Define new syntax so we can write something like
; [1 2 3 4 5 6]
;instead of
; '(1 2 3 4 5 6)
⊙ (defun open-bracket-macro-character (stream char)
     `',(read-delimited-list #\] stream t))
→ OPEN-BRACKET-MACRO-CHARACTER

⊙ (set-macro-character #\[ #'open-bracket-macro-character)
→ T

⊙ (set-macro-character #\] (get-macro-character #\)))
→ T

;Now try it:
⊙ [1 2 3 4 5 6]
→ (1 2 3 4 5 6)
```

First we tried to evaluate `(1 2 3 4 5 6)` -- this is wrong because `1` is not a function. What we really meant to do was to quote the list. But if we're going to do this often, we'd like a more convenient syntax. In particular we'd like `[...]` to behave like `'(...)`.

To make this work, we have to define a specialized list reader macro function that doesn't evaluate its arguments. We'll arrange for the function to be called when the reader encounters a `[` character; the function will return the list when it encounters a `]` character. Every reader macro function gets called with two arguments: the input stream and the character that caused the macro to be invoked.

Our reader macro is very simple because Lisp has a function designed to read delimited lists. `READ-DELIMITED-LIST` expects one argument -- the character which will terminate the list being read. The other two arguments are optional -- the input stream and a flag which is normally set to `T` when used in reader macro functions. `READ-DELIMITED-LIST` reads objects from the input stream until it encounters the terminating character, then returns all of the objects in a list. By itself, this does everything we need *except* for suppressing evaluation.

`QUOTE` (or `'`) suppresses evaluation, as we saw in Lesson 3. But we can't use `'(READ-DELIMITED-LIST ...)`; that would suppress evaluation of the form we need to evaluate to get the form we need to quote... Instead, we use ` (see Lesson 8) to selectively require evaluation within a quoted form.

Our definition of `OPEN-BRACKET-MACRO-CHARACTER` uses

```
`',form
```

to evaluate *form* and return the result, quoted.

Lisp reserves six characters for the programmer:

```
 [ ] { } ! ?
```

You can define any or all of these as macro characters without interfering with Lisp. However, you should watch out for conflicts if you share code with other programmers.

Contents | Cover
Chapter 2 | Chapter 3, Introduction | Chapter 3, Lesson 11 | Chapter 3, Lesson 12 | Chapter 4 |

Copyright © 1995-2001, David B. Lamkins
All Rights Reserved Worldwide

This book may not be reproduced without the written consent of its author. Online distribution is restricted to the author's site.

Chapter 3 - Essential Lisp in Twelve Lessons

This chapter will teach you everything you need to know to get started with Lisp. I'll cover all of the core features of the language. I encourage you to think of this core as the Lisp language itself, and everything else as a very large standard library. With this background, you'll be much better equipped to learn the rest of Lisp, either by reading the rest of the book or via a reference manual such as *Common Lisp: The Language, 2nd Edition*.

You should read this chapter all the way through. At times, I mention other chapters and later sections of this chapter, but you shouldn't have to follow these references to understand this chapter. When you finish this chapter, you should work through Chapter 4 while sitting at the keyboard of your Lisp system.

- Lesson 1 - Essential Syntax
- Lesson 2 - Essential Evaluation
- Lesson 3 - Some Examples of Special Forms and Macros
- Lesson 4 - Putting things together, and taking them apart
- Lesson 5 - Naming and Identity
- Lesson 6 - Binding versus Assignment
- Lesson 7 - Essential Function Definition
- Lesson 8 - Essential Macro Definition
- Lesson 9 - Essential Multiple Values
- Lesson 10 - A Preview of Other Data Type
- Lesson 11 - Essential Input and Output
- Lesson 12 - Essential Reader Macros

Contents | Cover
Chapter 2 | Chapter 3, Introduction | Chapter 4

Copyright © 1995-2001, David B. Lamkins
All Rights Reserved Worldwide

This book may not be reproduced without the written consent of its author. Online distribution is restricted to the author's site.

Chapter 4 - Mastering the Essentials

We've explored the fundamental concepts of Lisp through the twelve lessons of Chapter 3. If you feel that you have a very strong grasp of these fundamentals, or if you've worked with Lisp before, you may want to skim the remainder of this chapter.

We'll review some of the material from Chapter 3 using a hands-on approach. Along the way, you'll learn some new techniques that have had to wait until all of the fundamentals had been introduced; if you're a beginner and haven't read Chapter 3, go back and read it before you try to do the exercises in this chapter.

You should have access to a Lisp development system as you work through this chapter. As you read this chapter, please take the time to run the examples using your Lisp system. This will give you a chance to learn how your Lisp system responds to input, including any mistakes you may make. (If you don't make any mistakes in transcribing the examples, you should get adventurous and try to modify some of the examples.) Appendix A lists several commercial, shareware, and free Lisp systems for Macintosh, DOS, and Windows computers.

Hands-on! The "toploop"

You interact with the Lisp system through a built-in piece of code called the toploop, which repeats three simple steps for as long as you run the Lisp system:

```
1. Read an expression (you provide the expression).
2. Evaluate the expression just read.
3. Print the result(s) of the evaluation.
```

This is also called the "read-eval-print" loop. Some Lisp systems evaluate the expression using a Lisp interpreter; modern systems use a compiling evaluator, which first compiles the expression to machine code then executes the code. A compiling evaluator is also an incremental compiler, so named because it can compile a program in increments of one expression.

The toploop also provides a minimal user interface -- a prompt to indicate that it's ready to read a new expression -- and a way to gracefully catch any errors you might make.

If you were to write the Lisp code for a toploop, it would look something like this:

```
(loop
   (terpri)
   (princ 'ready>)
   (print (eval (read))))
```

> NOTE 1: (terpri) prints a blank line.
>
> NOTE 2: (loop ...) executes its forms in order, then repeats -- we'll see more of `LOOP` in Chapter 5.
>
> NOTE 3: (eval ...) returns the result of evaluating a form. This is one of the few legitimate uses of `EVAL` -- you should beware of Lisp code that uses `EVAL` for reasons other than evaluating arbitrary Lisp expressions provided at runtime.

In fact, you can type this into your Lisp system and temporarily run your own toploop *on top of* Lisp's toploop. Try it! You'll see your system's prompt replaced with `READY>`. Every valid Lisp form you type will be read, evaluated, and printed by *your* toploop. Depending upon your Lisp system, this may happen as soon as the expression is completed -- either by a space or a matching parenthesis or double quote mark -- or you may have to press the `RETURN` or `ENTER` key.

Your Lisp session may look like the following, where `?` is the Lisp system's prompt for input:

```
? (loop
     (terpri)
     (princ 'ready>)
     (print (eval (read))))

READY>(+ 1 2 3)

6
READY>(cons 1 (cons 2 (cons 3 nil)))

(1 2 3)
READY>
```

There are two ways to get out of your toploop. One is to *abort*, typically using a special keystroke or a menu command -- consult your Lisp system's manual. The other way is to enter an erroneous expression -- such as `(+ 'A 1)` -- at the `READY>` prompt, which will put you into the Lisp debugger.

In Lisp, the debugger is accessed via a "break loop." This behaves just like a toploop, but accepts additional commands to inspect or alter the state of the "broken" computation. Break loops vary widely among Lisp systems. The manual will describe the break loop. Check also under the manual's index entries for "debugger."

Spotting and avoiding common mistakes

"I entered a Lisp expression, but nothing happened." The most common cause of this problem is missing a matching delimiter -- typically a right parenthesis or double-quote -- somewhere in your expression. Unlike some development systems which process your input each time you enter a line of code, Lisp waits for you to enter a complete expression before attempting to process anything. What happens if you enter the following code in your system?

```
? (defun bad-1 ()
     (print "This is a bad function definition)
     (print "But I'll try it anyway..."))
```

Looks good, huh? All the parentheses match, and you press the `ENTER` key that one last time, and... Nothing. The string argument to the first print statement is missing a closing double-quote, turning the rest of your input into part of the string. You'll do this more than once (trust me), so the best thing to do is to consult your Lisp system manual to find out how to edit the pending input so you can add the missing double-quote to what you've already typed.

Here's another bit of code that will make your Lisp system appear to sleep:

```
? (defun factorial (n)
     (cond ((<= n 0)  1)
           (t (* n (factorial (- n 1))))))
```

Again, a quick glance shows nothing amiss. But count the parentheses and you'll find that lefts outnumber rights by one. When you press the final enter key, the read part of Lisp's read-eval-print loop still needs one more right parenthesis before it can finish its job and pass your expression to the evaluator.

Both of these situations can be avoided if your Lisp system has an editor that matches delimiters as you type. In some systems, this matching momentarily flashes the left delimiter as you type the matching right delimiter. Or your system might flash or highlight the matching delimiter of whatever you have under the cursor; some systems even highlight the entire intervening expression. Again, check your manual -- this feature is **essential** to comfortable Lisp programming.

"I get confused about when to use `'`." This is a really common problem for people just learning to program, but it manages to puzzle the occasional experienced (non-Lisp) programmer as well. The rule is simple:

> If you want a name to stand for a value, *don't* quote it.
>
> If you want a name to stand for its symbol, quote it.

There are a few exceptions to the rule, all having to do with self-evaluating symbols. These symbols always represent themselves. They are:

```
T
NIL
```

and *keyword* symbols. A keyword symbol is any symbol that begins with a `:` character, for reasons that will become clear when we look at *packages* in Chapter 31. A keyword symbol always evaluates to itself, thus:

```
? :foo
:FOO
? :some-long-but-nondescript-keyword-symbol
:SOME-LONG-BUT-NONDESCRIPT-KEYWORD-SYMBOL
```

It usually doesn't hurt to quote a self-evaluating symbol. For example, `NIL` is identical to `'NIL`. Adding the quote is a matter of style and preference.

Time for a pop quiz! What's wrong with the following code?

```
? (defun factorial (n)
     (cond ((<= 'n 0)  1)
           (t (* 'n (factorial (- 'n 1))))))
```

Right. The `'N` expressions are wrong, because we want the value of the symbol (a number which varies with execution of the function), and *not* the symbol itself.

Defining simple functions

We've already seen a few function definitions: the `FACTORIAL` function (above) and a function or two in Chapter 3, Lesson 7. To review, a function is defined as follows:

```
(defun function-name (argument-names ...)
   function-body )
```

The `(` *argument-names* `...)` is called a **lambda list**. Names in this list are bound to values when the function is called. The body of the function may refer to these names; identical names appearing elsewhere in your program (that is, outside the function body) are irrelevant to the function. Also, if your function changes the binding of an argument inside the function, the caller *does not* receive the changed value. The proper way to return values from a Lisp function is to return them as the value of the function.

For example:

```
? (defun quadratic-roots (a b c)
    "Returns the roots of a quadratic equation aX^2 + bX + c = 0"
    (let ((discriminant (- (* b b) (* 4 a c))))
      (values (/ (+ (- b) (sqrt discriminant)) (* 2 a))
              (/ (- (- b) (sqrt discriminant)) (* 2 a)))))
QUADRATIC-ROOTS

? (quadratic-roots 1 2 4)
#c(-1.0 1.7320508075688772)
#c(-1.0 -1.7320508075688772)

? (quadratic-roots 2 -16 36)
#c(4.0 1.4142135623730951)
#c(4.0 -1.4142135623730951)

? (quadratic-roots 1 4 4)
-2
-2

? (quadratic-roots 1 -14 49)
7
7

? (quadratic-roots 1 8 4)
-0.5358983848622456
-7.464101615137754

? (quadratic-roots 1 4 -5)
1
-5
```

The `QUADRATIC-ROOTS` function shows how to use a documentation string. The first form in the function body is a string. This does not affect the function result, but it *is* recorded by the Lisp system for later reference:

```
? (documentation 'quadratic-roots 'function)
"Returns the roots of a quadratic equation aX^2 + bX + c = 0"
```

This function also shows how we can return two values from a function. You recognize the formula for the roots of a quadratic equation:

```
             /--------
       +    /  2
   - b -  \/  b  - 4ac
 ----------------------
          2a
```

This tells you that the equation has two solutions (which may be coincident in some cases). In Lisp it's a simple matter to return both values at once using the form `(VALUES` *`value-1 value-2`*`)`.

If you've ever solved this problem in a computer language that doesn't support complex number arithmetic, you've had to find a way to signal the caller when the roots are imaginary (i.e. when the discriminant is less than zero). Lisp just does the right thing: the square root of a negative number is a complex number:

```
? (sqrt -1)
#c(0 1)
```

Suppose that you wanted `QUADRATIC-ROOTS` to only return one value if the roots are coincident. Thinking that maybe you can return *something* special as the second value of the `VALUE` form, you try `NIL` :

```
? (values 2 nil)
2
NIL
```

But that doesn't work, because `NIL` is a value like any other in Lisp, and does not get special treatment like a nil pointer would, for example, in another language.

So you think about only having one value in the `VALUES` form:

```
? (values 3)
3
```

Sure enough, that works. So why not `(VALUES` *`value-1 some-form-that-returns-nothing`*`)`? Like this:

```
? (values)

? (values 4 (values))
4
NIL
```

Unfortunately, this doesn't do what you expect; the outer `VALUES` form expects a value from its second argument, `(VALUES)`, and substitutes `NIL` for the missing value. This is one of the important rules of multiple values. The other rule is that forms which receive multiple values (see Chapter 3, Lesson 9) substitute `NIL` for a missing value.

A little reflection convinces you that you can't get `VALUES` to return nothing for something, so you consider having two separate returns. This yields the following function:

```
? (defun quadratic-roots-2 (a b c)
    "Returns the roots of a quadratic equation aX^2 + bX + c = 0.
Returns only one value if the roots are coincident."
    (let ((discriminant (- (* b b) (* 4 a c))))   ; zero if one root
      (cond ((zerop discriminant)
             ;; coincident roots -- return one value
             (/ (+ (- b) (sqrt discriminant)) (* 2 a)))
            (t
             ;; two distinct roots
            (values (/ (+ (- b) (sqrt discriminant)) (* 2 a))
                    (/ (- (- b) (sqrt discriminant)) (* 2 a)))))))
QUADRATIC-ROOTS-2

? (quadratic-roots-2 1 -14 49)
7

? (quadratic-roots-2 1 4 -5)
1
-5
```

> NOTE: `ZEROP` is a *predicate* (hence the `P` suffix) that is true when its numeric argument is zero. Writing `(ZEROP n)` is the same as writing `(= n 0)`.

Note how `QUADRATIC-ROOTS-2` has a two-line documentation string. The newline is part of the string:

```
? (documentation 'quadratic-roots-2 'function)
"Returns the roots of a quadratic equation aX^2 + bX + c = 0.
Returns only one value if the roots are coincident."
```

Also note the use of comments to describe the two return choices. In Lisp, a comment begins with a semicolon and continues until the end of the line. By convention, comments on a line of their own within a function body are indented to the same level as the rest of the code and prefixed by two semicolons. A comment on the same line as code only has one semicolon (again, by convention).

A lambda list can have a number of additional features. We'll look at two of these here, and the rest in Chapter 21.

If you want to make a function that takes one or more optional arguments, use the `&OPTIONAL` keyword followed by a list of parameter names, like this:

```
? (defun silly-list-1 (p1 p2 &optional p3 p4)
    (list p1 p2 p3 p4))
SILLY-LIST-1

? (silly-list-1 'foo 'bar)
(FOO BAR NIL NIL)

? (silly-list-1 'foo 'bar 'baz)
(FOO BAR BAZ NIL)

? (silly-list-1 'foo 'bar 'baz 'rux)
(FOO BAR BAZ RUX)
```

The optional parameters default to NIL when the call does not supply a value. Peek ahead to Chapter 21 to see how to change the default value of an optional parameter.

If you supply fewer than the number of required parameters (to the left of `&OPTIONAL` in the example above), or more than the total number of required plus optional parameters, you'll get an error:

```
? (silly-list-1 'foo)
Error: Not enough arguments.
```

```
? (silly-list-1 'foo 'bar 'baz 'rux 'qup)
Error: Too many arguments.
```

If you want to have an indefinite number of parameters, you can name one parameter to receive a list of all the "extras" using the `&REST` symbol in the lambda list, like this:

```
? (defun silly-list-2 (p1 p2 &rest p3)
    (list p1 p2 p3))

? (silly-list-2 'foo 'bar)
(FOO BAR NIL)

? (silly-list-2 'foo 'bar 'baz)
(FOO BAR (BAZ))

? (silly-list-2 'foo 'bar 'baz 'bob 'tom 'don)
(FOO BAR (BAZ BOB TOM DON))
```

The `&REST` parameter must follow all of the required parameters. You can combine `&REST` and `&OPTIONAL` parameters, observing the following order:

```
? (defun silly-list-3 (p1 p2 &optional p3 p4 &rest p5)
    (list p1 p2 p3 p4 p5))
SILLY-LIST-3

? (silly-list-3 'foo 'bar)
(FOO BAR NIL NIL NIL)

? (silly-list-3 'foo 'bar 'baz)
(FOO BAR BAZ NIL NIL)

? (silly-list-3 'foo 'bar 'baz 'bob)
(FOO BAR BAZ BOB NIL)

? (silly-list-3 'foo 'bar 'baz 'bob 'tom)
(FOO BAR BAZ BOB (TOM))

? (silly-list-3 'foo 'bar 'baz 'bob 'tom 'don)
(FOO BAR BAZ BOB (TOM DON))
```

Using global variables and constants

In Lesson 3, we used `SETQ` to define global variables. You can do this using a top-level form, as in Lesson 3, or from within a function, such as this:

```
? (defun set-foo-globally (x)
    (setq foo x))
SET-FOO-GLOBALLY

? foo
Error: unbound variable FOO

? (set-foo-globally 3)
3
```

```
? foo
3
```

Depending upon your Lisp system, you may have seen a warning message when you defined `SET-FOO-GLOBALLY`:

```
? (defun set-foo-globally (x)
    (setq foo x))
Warning: undeclared free variable FOO, in SET-FOO-GLOBALLY.
SET-FOO-GLOBALLY
```

This is not an error -- the function does what we want. But `FOO` is said to be free because the function does not create a binding for `FOO`. Variable bindings are created by lambda lists (the function's argument list) and by `LET` forms (see Lesson 6), among others.

My Lisp system warns me about free variables in function definitions because they could be a symptom of a typographical error:

```
? (setq *olympic-year* 1996)
1996

? (defun set-next-olympic-year ()
    (setq *olympic-year* (+ *olmpic-year* 2)))
Warning: undeclared free variable *OLMPIC-YEAR*, in SET-NEXT-OLYMPIC-YEAR.
SET-NEXT-OLYMPIC-YEAR
```

Here, I misspelled the second instance of my global variable `*OLYMPIC-YEAR*`, and the compiler warned me. Notice that I didn't get a warning for the correctly spelled `*OLYMPIC-YEAR*` because I had defined it globally in a top-level `SETQ` form.

There are two more ways to define global variables in Lisp:

```
? *var1*
Error: unbound variable

? (defvar *var1* 1)
*VAR1*

? *var1*
1

? (defvar *var1* 2)
*VAR1*

? *var1*
1

? (defparameter *a-var* 3)
*A-VAR*

? *a-var*
3

? (defparameter *a-var* 4)
```

```
*A-VAR*

? *a-var*
4
```

`DEFVAR` sets a global value only the first time -- in other words, the variable must not have a value in order for `DEFVAR` to have an effect. This is useful for a variable that needs to have an initial value, but shouldn't be reset if you re-evaluate the `DEFVAR` form (as you might if you reload the file containing the `DEFVAR` in addition to other code).

`DEFPARAMETER` sets a global value each time it is used. Although the effect is the same as a `SETQ` form, the `DEFPARAMETER` is preferable because it gives implicit documentation as a *defining form* (in Lisp, any form that begins with `DEF` is most likely a defining form), and because it allows you to add documentation to the variable:

```
? (defparameter *a-var* 3 "The number of things I have to do today.")
*A-VAR*

? (documentation '*a-var* 'variable)
"The number of things I have to do today."
```

You can also add a documentation string to a `DEFVAR` form.

In the examples above, we've started following the convention of making global variable names begin and end with an asterisk. When you read other programmers' Lisp code, you'll see that they follow this convention. They'll expect you to do the same.

`DEFCONSTANT` is similar to `DEFVAR` and `DEFPARAMETER`, except that it defines a name which is known globally and has a *constant* value. This means that anywhere you read a name which was defined in a `DEFCONSTANT` form, you can substitute the value given by the `DEFCONSTANT` form. It also means that you can't redefine the named constant, not even by using another `DEFCONSTANT` form with a different value.

Some Lisp programmers give constants names which begin and end with plus signs. It's helpful to name constants in a distinctive way so you don't inadvertently try to use the name for another purpose. Once a name has been defined constant, you can't even use it for a seemingly innocuous use, such as a parameter in a lambda list or `LET` binding.

Defining recursive functions

A function that calls itself is *recursive*. The recursive call may be direct (the function calls itself) or indirect (the function calls another function which -- perhaps after calling still more functions -- calls the original function).

You need to follow two simple rules of thumb to make recursive functions work. These rules suggest the structure of a recursive function -- it must behave appropriately according to its current inputs:

1. One case must *not* make a recursive call.
2. Other cases must *reduce* the amount of work to be done in a recursive call.

Let's dig up the `FACTORIAL` function that we've already used in several examples, and see how it follows these rules:

```
(defun factorial (n)
  (cond ((zerop n) 1)
        (t (* n (factorial (1- n))))))
```

This function has two cases, corresponding to the two branches of the `COND`. The first case says that the factorial of zero is just one -- no recursive call is needed. The second case says that the factorial of some number is the number multiplied by the factorial of one less than the number -- this is a recursive call which reduces the amount of work remaining because it brings the number closer to the terminating condition of the first `COND` clause. (For clarity, I've assumed that the number initially given to `FACTORIAL` is non-negative.)

Let's work through another simple recursive definition. The length of an empty list is zero. The length of a non-empty list is one plus the length of the list reduced by one element. These two statements state exactly what is required by our rules of thumb, above. The first statement gives the answer for a list of known length -- the trivial case of an empty list. The second statement gives the answer for a list of unknown length *in terms of the answer for a list of reduced length*. Here's how it translates into code:

```
? (defun my-length (list)
    (cond ((null list) 0)
          (t (1+ (my-length (rest list))))))
MY-LENGTH

? (my-length '(a b c d))
4
```

`NULL` is true for an empty list, so the first `COND` clause returns zero for the empty list. The second `COND` clause gets evaluated (if the first clause if skipped) because its condition is `T`; it adds one to the result of the recursive call on a list which is one element shorter (a list consists of its `FIRST` element and the `REST` of the list.)

Note the similarities between `FACTORIAL` and `MY-LENGTH`. The base case is always the first in the `COND` because it must be tested *before* the recursive case -- otherwise, the recursive function calls would never end.

If you want to visualize how recursive calls work, you can use you Lisp system's `TRACE` macro:

```
? (trace my-length)
NIL

? (my-length '(a b c d))
; Calling (MY-LENGTH (A B C D))
;  Calling (MY-LENGTH (B C D))
;   Calling (MY-LENGTH (C D))
;    Calling (MY-LENGTH (D))
;     Calling (MY-LENGTH NIL)
;     MY-LENGTH returned 0
;    MY-LENGTH returned 1
;   MY-LENGTH returned 2
;  MY-LENGTH returned 3
; MY-LENGTH returned 4
4
```

Here, you can clearly see the recursive calls upon lists of decreasing length, the terminating call with the empty list (`NIL`), and the returns each adding one to the length of a shorter list.

> NOTE: Your Lisp compiler may internally optimize the recursive calls to `MY-LENGTH` so you don't see them using `TRACE`. If this happens, you may be able to disable the optimization by evaluating the form `(DECLAIM (OPTIMIZE (SPEED 0) (DEBUG 3)))`, then re-evaluating the `(DEFUN MY-LIST ...)` form.

Tail recursion

A function that calls itself as its very last action is said to make a tail-recursive call. Here are two versions of the factorial function to illustrate the difference between a tail-recursive call and an ordinary recusive call:

```
; Normal recursive call

(defun factorial (n)
  (cond ((zerop n) 1)
        (t (*      ; * is the last function called
            n
            (factorial (- n 1))))))

; Tail-recursive call

(defun factorial-tr (n)
  (factorial-tr-helper n 1))

(defun factorial-tr-helper (n product)
  (cond ((zerop n) product)
        (t
         ; factorial-tr-helper is the last function called
         (factorial-tr-helper (- n 1) (* product n)))))
```

`FACTORIAL-TR` calls `FACTORIAL-TR-HELPER`, passing the original argument, `N`, plus an additional argument used as the initial value of an accumulator for the product which will become the value of the factorial calculation. `FACTORIAL-TR-HELPER` calls itself recursively, decrementing `N` in the process (this moves the calculation closer to its terminating condition, `(ZEROP N)`) and at the same time multiplying the product by the current value of `N`.

Because `FACTORIAL-TR-HELPER` is the last function executed in the recursive call, this is a tail-recursive call. Compare this to the recursive call in the `FACTORIAL` function, where the result is used by `*` to produce the function's value. A recursive call is tail-recursive only if it is the very last function executed in the recursive invocation.

With all that explanation out of the way, you're probably wondering "What good is tail-recursion? For the factorial calculation, it only seemed to complicate the code." The answer is in two parts: what Lisp can do *for* you, and what Lisp can do *to* you in the presence of tail-recursion.

Some Lisp compilers can optimize tail-recursive calls. To understand the benefits of such an optimization, let's first look at what a compiler must do for a normal function call: it must generate code to evaluate the arguments and push them on a stack (where they can be found by the called function), save the address in the code to which control will return after the recursive call, and finally call the function. One implication of this code sequence is that a function which makes a lot of recursive calls (as `FACTORIAL` will do for large value of `N`) will use a lot of stack space -- normally a limited resource.

A compiler that optimizes tail-recursive calls will generate code to perform the following operations for a tail-recursive call: evaluate the arguments and replace the old argument values with those just calculated, and then jump to the beginning of the function. Note that this code does not use any additional stack space, and it invokes the function with a jump instead of a call instruction -- this is a less expensive operation on all computers.

So, that's the answer to the first question, "What can Lisp do *for* me if I write a tail-recursive function call?" You get more efficient code -- **if** the compiler performs that optimization; it is not required to do so, but the better ones do.

Tail recursion optimization sounds like a good thing. It must be -- it produces faster code -- but it it may confuse you during debugging. The debugger normally displays each function call by looking at the stack frame created at entry to the function. So if you happen to break in the middle of a recursive function, you'd expect to see a stack frame for each recursive call:

```
? (defun broken-factorial (n)
    (cond ((= n 0) 1)
          ((= n 1) (break))
          (t (* n (broken-factorial (- n 1))))))
BROKEN-FACTORIAL

? (broken-factorial 6)
; Break: While executing: BROKEN-FACTORIAL

> (backtrace)
1: (BROKEN-FACTORIAL 1)
2: (BROKEN-FACTORIAL 2)
3: (BROKEN-FACTORIAL 3)
4: (BROKEN-FACTORIAL 4)
5: (BROKEN-FACTORIAL 5)
6: (BROKEN-FACTORIAL 6)
7: ... more stack frames, unrelated to BROKEN-FACTORIAL ...
> (abort)
; Return to top level

? (defun broken-tr-factorial (n)
    (broken-tr-factorial-1 n 1))
BROKEN-TR-FACTORIAL

? (defun broken-tr-factorial-1 (n v)
    (cond ((= n 0) v)
          ((= n 1) (break))
          (t (broken-tr-factorial-1 (- n 1) (* n v)))))
BROKEN-TR-FACTORIAL

? (broken-tr-factorial 6)
; Break: While executing: BROKEN-TR-FACTORIAL-1

> (backtrace)
1: (broken-tr-factorial-1 1)
2: ... more stack frames, unrelated to BROKEN-TR-FACTORIAL ...
```

So what happened to all the recursive calls in `BROKEN-TR-FACTORIAL-1`? For that matter, what happened to the call to `BROKEN-TR-FACTORIAL`? The compiler did tail recursion elimination in `BROKEN-TR-FACTORIAL-1`, replacing function calls with jumps. The function only generated one stack frame, then the tail-recursive calls replaced the values in that frame for subsequent calls.

The compiler also noticed that `BROKEN-TR-FACTORIAL` calls `BROKEN-TR-FACTORIAL-1` and immediately returns its value. This is just another tail-recursive call. The compiler arranged to build the stack frame using the value provided for the call to `BROKEN-TR-FACTORIAL` and the constant 1; there was no need to generate a stack frame for `BROKEN-TR-FACTORIAL`.

I mention all of this because you may think that your compiler is broken the first time you encounter a backtrace with "missing" frames. Compilers that do tail recursion usually give you a way to disable that optimization; consult the manual for details. You're probably better off, however, learning to recognize tail recursion, and how to read backtraces in the presence of this optimization. Some code which relies on tail recursion could break (by overflowing the stack) if you disable the optimization.

Exercises in naming

A name in Lisp can be made of any non-whitespace characters except for certain characters reserved as reader macro characters (see Chapter 3, Lesson 11), namely `"`, `'`, `(`, `)`, `,`, `;`, `` ` ``, and `#`. Furthermore, the name can't be a number in the current number base, as set by `*READ-BASE*`. Thus, FACE is a name when `*READ-BASE*` is 10, but a number when `*READ-BASE*` is 16 (or higher).

Most Lisp programmers follow a few naming conventions to identify the names that certain roles. Global variables are almost always written with a leading and trailing `*`, for example:

```
*next-id*
*home-directory*
*software-version*
```

Other conventions vary somewhat among Lisp programmers. It is fairly common to see the name of a constant written with a leading and trailing `+`, such as:

```
+initial-allocation-count+
+maximum-iteration-limit+
```

However, Lisp itself does not follow this convention for constants defined by the language:

```
pi
most-positive-fixnum
least-negative-short-float
```

Lisp programmers tend to set aside certain characters as prefixes for names of functions which use implementation-dependent features of the Lisp implementation, or which which are otherwise considered "dangerous" because they violate abstraction. The `%` character is most often seen in this role, but others are used -- you should be aware that any name which starts with a non-alphabetic character *may* have some special significance to the programmer who wrote the code:

```
%open-file-id
%structure-slot-names
$reserve_heap
_call-event-handler
@frame-marker
```

Don't forget to use the proper forms (described earlier in this chapter) to declare global variables and constants. Many Lisp compilers will let you get away with using a SETQ form to define global variables. Although this is convenient for debugging purposes, you should not rely on this behavior in your final program, as it is not guaranteed to work in all implementations.

If you don't define a constant using a `DEFCONSTANT` form, the compiler can not guarantee that its value will remain constant. Even worse is the requirement that a constant name be neither assigned (through a `SETQ` form, for example) nor bound (in a `LET` form or as the name of a function parameter, for example). If you don't define your constants using `DEFCONSTANT`, the compiler has no way to enforce these requirements.

Lexical binding, and multiple name spaces

The following piece of code illustrates how you can use the same name for different purposes. Take a minute to read this, and see how many separate uses you can count for the name `FUNNY`.

```
(defun funny (funny)
  "funny..."
  (if (zerop funny)
    :funny
    (list
     (cons funny
           (let ((funny funny))
             (setq funny (1- funny))
             (funny funny)))
     funny)))
```

Here are the five roles played by this one name:

1. function name
2. function argument
3. a word in the documentation string
4. a constant in the keyword package
5. a new lexical variable

Considering only the symbols named `FUNNY`, there are different values according to its use and position in the code. First, there is its value as a function object -- this is created by the `DEFUN` form and called recursively inside the `LET` form. Next, the value of the actual parameter is passed in a call to the function and bound to this name. Then, there's the constant value of the keyword, appearing as the consequent return value of the `IF` form. And finally, inside the `LET` form, a new binding is created (by the `LET` form) and its value changed (by the `SETQ` form).

Is this hard to follow? Yes. As a rule of thumb, you should be shot if you write code that looks like this. I, on the other hand, get to do this because it's instructive -- the lesson here is that there are a number of different namespaces in Lisp.

And what happens when you invoke this bizarre function? This:

```
? (funny 3)
((3 (2 (1 . :FUNNY) 1) 2) 3)

? (funny 0)
:funny
```

Now consider the following Lisp session:

```
? (defun foo () 1)
FOO

? (defun baz ()
    (flet ((foo () 2)
           (bar () (foo)))
      (values (foo) (bar))))
BAZ

? (baz)
2
1

? (defun raz ()
    (labels ((foo () 2)
             (bar () (foo)))
      (values (foo) (bar))))
RAZ

? (raz)
2
2
```

This is pretty subtle, but it's worth understanding because this is fairly common practice. Here's what happened:

1. define function `FOO` to return 1
2. define function `BAZ`, which
    1. defines function `FOO` locally to return 2
    2. defines function `BAR` locally to call `FOO`
    3. calls `FOO` and `BAR`, and returns their values
3. call `BAZ`, which returns the values 2 and 1
4. define function `RAZ`, which
    1. defines function `FOO` locally to return 2
    2. defines function `BAR` locally to call `FOO`
    3. calls `FOO` and `BAR`, and returns their values
5. call `RAZ`, which returns the values 2 and 2

Even though `BAZ` and `RAZ` ostensibly do the same thing, they return different values.

`BAZ` defines its local functions inside an `FLET` form, which does not allow reference to the functions it defines. So the `FOO` called by `BAR` inside `BAZ` is actually the global `FOO`, which returns 1. The `FOO` defined inside the `FLET` form is never referenced by `BAZ`.

`RAZ` defines its local functions inside a `LABELS` form, within which functions defined may refer to themselves or each other. Thus, the `FOO` called by `BAR` inside `RAZ` is the one defined inside the `LABELS` form, which returns 2. The globally defined `FOO` is shadowed by the `FOO` named in the `LABELS` form.

In both cases, `FOO` is lexically apparent at two places: globally, and within the local defining form (`FLET` or `LABELS`). For something to be lexically apparent, or lexically scoped, means that its definition can be determined by reading the program.

In `BAZ`, I know that the local definition of `FOO` is not visible within `BAR`, so the global definition must be referenced. (If there was an enclosing form within `BAZ` which defined a local function `FOO`, that would be referenced rather than the global definition -- again, because it's lexically apparent to the caller.)

In `RAZ`, I know that the local definition of `FOO` is visible to `BAR`, so this is used instead of the global definition. Even if there was an enclosing form that defined another `FOO` locally within `RAZ`, it would -- from the viewpoint of `BAR` -- be shadowed by the `FOO` defined in the `LABELS` form.

Reading, writing, and arithmetic

Your programs usually need to get input and produce output. If you're working with a system that supports windows and dialogs, you can certainly use these graphical devices. Relying instead on Lisp's built-in facilities for reading and writing strings of characters will ensure that your program is useful (or at least usable) on all kinds of computers.

Most elementary programming texts include a simple program to demonstrate the "input, process, output" approach. Our example in Lisp reads a series of numbers, adds them, and prints the sum when we enter a special token instead of a number:

```
(defun simple-adding-machine-1 ()
  (let ((sum 0)
        next)
    (loop
      (setq next (read))
      (cond ((numberp next)
             (incf sum next))
            ((eq '= next)
             (print sum)
             (return))
            (t
             (format t "~&~A ignored!~%" next))))
    (values)))
```

Our `SIMPLE-ADDING-MACHINE-1` works like this:

```
(SIMPLE-ADDING-MACHINE-1)
3
5
FOO
FOO ignored!
11
=

19
```

`SIMPLE-ADDING-MACHINE-1` gets its input via the keyboard, and writes output to the screen. This happens because `READ` and `PRINT` have optional arguments which specify a *stream* (see Chapter 19) and because using `T` as the second argument to `FORMAT` is the same as specifying the standard output stream -- the screen.

What if we wanted to read inputs from a file, and write to another file? One way is to bind the standard input and output streams to files, and continue to use `SIMPLE-ADDING-MACHINE-1`:

```
(let ((*standard-input* (open "infile.dat" :direction :input))
      (*standard-output* (open "outfile.dat" :direction :output)))
  (declare (special *standard-input* *standard-output*))
  (simple-adding-machine-1)
  (close *standard-input*)
  (close *standard-output))
```

This is almost, but not quite, satisfactory. We bind the standard input and output streams to newly opened files, process the data, and close the files. We use `LET` to temporarily bind the standard streams to files; upon leaving the `LET` form, `*STANDARD-INPUT*` and `*STANDARD-OUTPUT*` regain their original values. The problem lurking in this piece of code is that an abnormal exit -- an error or a deliberate interrupt -- can cause one or both of the `CLOSE` calls to be skipped.

A better way to write this kind of code uses `WITH-OPEN-FILE`:

```
(with-open-file (in-stream "infile.dat" :direction :input)
  (with-open-file (out-stream "outfile.dat" :direction :output)
    (let ((*standard-input* in-stream)
          (*standard-output* out-stream))
      (declare (special *standard-input* *standard-output*))
      (simple-adding-machine-1))))
```

This does exactly the same thing, except that a file opened by `WITH-OPEN-FILE` is guaranteed to be closed upon exiting the form, whether the exit is normal or not. We'll take a look at how this is possible in Chapter 9.

The technique of rebinding the standard input and output streams can be very handy if you have to redirect input and output for a program you didn't write, don't want to rewrite, or can't get the source code to. If you're writing a program from scratch, you might want to plan for it to be used either with the standard streams or streams (perhaps attached to files) provided by the caller:

```
(defun simple-adding-machine-2 (&optional (in-stream *standard-input*)
                                          (out-stream *standard-output*))
  (let ((sum 0)
        next)
    (loop
      (setq next (read in-stream))
      (cond ((numberp next)
             (incf sum next))
            ((eq '= next)
             (print sum out-stream)
             (return))
            (t
             (format out-stream "~&~A ignored!~%" next))))
    (values)))
```

If you want to use `SIMPLE-ADDING-MACHINE-2` with the keyboard and screen, call it without any arguments. To call it with file streams, do this:

```
(with-open-file (in-stream "infile.dat" :direction :input)
  (with-open-file (out-stream "outfile.dat" :direction :output)
    (simple-adding-machine-2 in-stream out-stream)))
```

We don't have to rebind the standard input and output streams as we did to redirect I/O for `SIMPLE-ADDING-MACHINE-1`. This leaves the standard streams free other purposes -- such as reporting progress or interacting with the user.

To close out this section, let's take a brief look at arithmetic. Lisp has an extensive repertoire of mathematical functions, consult a reference book for details. Chapter 3, Lesson 10 covered numbers very briefly. Now, we're going to look at how and when numbers get converted automatically from one type to another.

The simplest rule is that of *floating point contagion*, an ominous-sounding term which means, "If you use a floating point number in a calculation, the result will be a floating point number."

The next rule involves floating point components of complex numbers. A complex number has a real part and an imaginary part, read (and printed) by Lisp as `#C(`*real-part imaginary-part*`)`, where *real-part* and *imaginary-part* are any kind of Lisp number except for another complex number. If either part is a floating point number, then Lisp converts both parts to floating point numbers.

If you reduce the imaginary part of a complex number to zero, you get the non-complex value of the real part.

Ratios are read and printed as *numerator*`/`*denominator*, where *numerator* and *denominator* are always integers. The advantage of a ratio is that it is exact -- `(/ 1.0 3)` is a floating point number which is very close to (but not exactly) one-third, but `1/3` (or `(/ 1 3)`) is *exactly* one-third.

A ratio whose numerator is exactly divisible by its denominator will be reduced to an integer -- again, this is an exact number.

And finally, an integer is just an integer. If an integer gets too large to fit the machine's representation, Lisp converts it to a *bignum* -- the number of digits is limited only by the computer's memory.

Just to make sure you understand all of this, try adding some numbers of different types to see whether you can invoke all of the conversions described above.

Other data types

Let's put together an extended example to show how we might use several of Lisp's built-in data types. We'll build a simple application to keep track of bank checks as we write them. For each check, we'll track the check number, payee, date, amount, and memo. We'll support queries to display an individual check, to list all checks paid to a payee, to list all the payees, to sum all of the check amounts, and to list all of the checks we've paid. We'll also provide a way to void a check once written.

Here's the code:

```
(defvar *checks* (make-array 100 :adjustable t :fill-pointer 0)
  "A vector of checks.")

(defconstant +first-check-number+ 100
  "The number of the first check.")

(defvar *next-check-number* +first-check-number+
  "The number of the next check.")

(defvar *payees* (make-hash-table :test #'equal)
  "Payees with checks paid to each.")

(defstruct check
  number date amount payee memo)

(defun current-date-string ()
  "Returns current date as a string."
  (multiple-value-bind (sec min hr day mon yr dow dst-p tz)
                       (get-decoded-time)
    (declare (ignore sec min hr dow dst-p tz))
    (format nil "~A-~A-~A" yr mon day)))

(defun write-check (amount payee memo)
  "Writes the next check in sequence."
  (let ((new-check (make-check
                    :number *next-check-number*
                    :date (current-date-string)
                    :amount amount
                    :payee payee
                    :memo memo)))
    (incf *next-check-number*)
    (vector-push-extend new-check *checks*)
    (push new-check (gethash payee *payees*))
    new-check))

(defun get-check (number)
  "Returns a check given its number, or NIL if no such check."
  (when (and (<= +first-check-number+ number) (<number *next-check-number*))
    (aref *checks* (- number +first-check-number+))))

(defun void-check (number)
  "Voids a check and returns T.  Returns NIL if no such check."
  (let ((check (get-check number)))
    (when check
      (setf (gethash (check-payee check) *payees*)
            (delete check (gethash (check-payee check) *payees*)))
      (setf (aref *checks* (- number +first-check-number+)) nil)
      t)))

(defun list-checks (payee)
  "Lists all of the checks written to payee."
  (gethash payee *payees*))

(defun list-all-checks ()
  "Lists all checks written."
  (coerce *checks* 'list))

(defun sum-checks ()
  (let ((sum 0))
    (map nil #'(lambda (check)
                 (when check
                   (incf sum (check-amount check))))
         *checks*)
    sum))

(defun list-payees ()
  "Lists all payees."
  (let ((payees ()))
    (maphash #'(lambda (key value)
                 (declare (ignore value))
                 (push key payees))
             *payees*)
    payees))
```

And here's an example of how it works:

```
? (write-check 100.00 "Acme" "T-1000 rocket booster")
```

```
#S(CHECK :NUMBER 100 :DATE "1996-11-3" :AMOUNT 100.0 :PAYEE "Acme" :MEMO "T-1000 rocket booster")

? (write-check 50.00 "Acme" "1 gross bungee cords")
#S(CHECK :NUMBER 101 :DATE "1996-11-3" :AMOUNT 50.0 :PAYEE "Acme" :MEMO "1 gross bungee cords")

? (write-check 300.72 "WB Infirmary" "body cast")
#S(CHECK :NUMBER 102 :DATE "1996-11-3" :AMOUNT 300.72 :PAYEE "WB Infirmary" :MEMO "body cast")

? (list-checks "Acme")
(#S(CHECK :NUMBER 101 :DATE "1996-11-3" :AMOUNT 50.0 :PAYEE "Acme" :MEMO "1 gross bungee cords")
 #S(CHECK :NUMBER 100 :DATE "1996-11-3" :AMOUNT 100.0 :PAYEE "Acme" :MEMO "T-1000 rocket booster"))
T

? (get-check 101)
#S(CHECK :NUMBER 101 :DATE "1996-11-3" :AMOUNT 50.0 :PAYEE "Acme" :MEMO "1 gross bungee cords")

? (sum-checks)
450.72

? (list-all-checks)
(#S(CHECK :NUMBER 100 :DATE "1996-11-3" :AMOUNT 100.0 :PAYEE "Acme" :MEMO "T-1000 rocket booster")
 #S(CHECK :NUMBER 101 :DATE "1996-11-3" :AMOUNT 50.0 :PAYEE "Acme" :MEMO "1 gross bungee cords")
 #S(CHECK :NUMBER 102 :DATE "1996-11-3" :AMOUNT 300.72 :PAYEE "WB Infirmary" :MEMO "body cast"))

? (list-payees)
("WB Infirmary" "Acme")

? (void-check 101)
T

? (list-checks "Acme")
(#S(CHECK :NUMBER 100 :DATE "1996-11-3" :AMOUNT 100.0 :PAYEE "Acme" :MEMO "T-1000 rocket booster"))
T

? (list-all-checks)
(#S(CHECK :NUMBER 100 :DATE "1996-11-3" :AMOUNT 100.0 :PAYEE "Acme" :MEMO "T-1000 rocket booster")
 NIL
 #S(CHECK :NUMBER 102 :DATE "1996-11-3" :AMOUNT 300.72 :PAYEE "WB Infirmary" :MEMO "body cast"))

? (sum-checks)
400.72
```

In about a page of code, we've built a simple check-writing application with efficient data structures to store checks and payees. We also have basic I/O facilities without any additional effort on our part. And thanks to garbage collection, we don't have to worry at all about storage deallocation or memory leaks.

Simple macros

The one important feature missing from our check writing program is the ability to save and restore its state. Since the state is completely contained in three global variables, `*CHECKS*`, `*NEXT-CHECK-NUMBER*`, and `*PAYEES*`, all we really have to do is to use `PRINT` to write the values of these variables to a file, and `READ` to reload them at a later time.

But with a little more work we can write a macro that will write our save and restore functions. Then we can use this macro not only for our check writing program, but also for any program which keeps its state in global variables.

First take a look at the finished macro, then we'll dissect it:

```
(defmacro def-i/o (writer-name reader-name (&rest vars))
  (let ((file-name (gensym))
        (var (gensym))
        (stream (gensym)))
    `(progn
       (defun ,writer-name (,file-name)
         (with-open-file (,stream ,file-name
                                  :direction :output :if-exists :supersede)
           (dolist (,var (list ,@vars))
             (declare (special ,@vars))
             (print ,var ,stream))))
       (defun ,reader-name (,file-name)
         (with-open-file (,stream ,file-name
                                  :direction :input :if-does-not-exist :error)
           (dolist (,var ',vars)
             (set ,var (read ,stream)))))
       t)))
```

The initial `LET` form defines symbols that will appear in the expanded macro. Each symbol is created by `(GENSYM)` so that no other symbol can possibly be the same. This avoids a problem which could arise if we wrote a macro using a particular symbol as a variable, then called the macro with an argument having the same name as one of the symbols in the expansion.

The expanded macro is generated by the ` form. The form is returned as the macro's expansion, then evaluated. Substitutions take place for symbols following `,` or `,@`. Everything else appears literally in the expanded macro.

The expansion of `DEF-I/O` is a `PROGN` form containing two `DEFUN` forms. We wrap the `DEFUN`s like this because a macro's expansion can only be a single form, and we need to have this macro define two functions.

The macro defines a writer function which loops over the list of the `VARS` specified in the macro call, printing each in turn to a named output file. The reader function loops over the *names* of the `VARS`, reading values from an input file and assigning the values to the named variables. Note that `SET` evaluates its first argument; this lets us use a variable to contain the name of the variable to which we want to assign a value.

Here's how the macro expands to create load and save functions for our check writer program:

```
? (pprint (macroexpand '(def-i/o save-checks load-checks (*checks* *next-check-number* *payees*
(PROGN (DEFUN SAVE-CHECKS (#:G2655)
            (WITH-OPEN-FILE (#:G2657 #:G2655 :DIRECTION :OUTPUT :IF-EXISTS :SUPERSEDE)
              (DOLIST (#:G2656 (LIST *CHECKS* *NEXT-CHECK-NUMBER* *PAYEES*))
                (DECLARE (SPECIAL *CHECKS* *NEXT-CHECK-NUMBER* *PAYEES*))
                (PRINT #:G2656 #:G2657))))
          (DEFUN LOAD-CHECKS (#:G2655)
            (WITH-OPEN-FILE (#:G2657 #:G2655 :DIRECTION :INPUT :IF-DOES-NOT-EXIST
                             :ERROR)
              (DOLIST (#:G2656 '(*CHECKS* *NEXT-CHECK-NUMBER* *PAYEES*))
                (SET #:G2656 (READ #:G2657))))))
```

And here's how we would use the macro, and the functions it defines, to save and restore the state information for our program:

```
? (def-i/o save-checks load-checks (*checks* *next-check-number* *payees*))
T
```

```
? (save-checks "checks.dat")
NIL

? (makunbound '*checks*)
*CHECKS*

? (makunbound '*next-check-number*)
*NEXT-CHECK-NUMBER*

? (makunbound '*payees*)
*PAYEES*

? *PAYEES*
Error: Unbound variable.

? (load-checks "checks.dat")
NIL

? *PAYEES*
("WB Infirmary" "Acme")
```

Reader macros

Our check-writing application has one small problem. If we use floating point numbers to represent dollars and cents, our sums could be off by a penny in some cases. What we should really do is to represent all currency in terms of whole pennies. We can make a reader macro to help with the input of dollar and cent amounts, converting input like `$10.95` into the corresponding number of pennies.

Here's the code:

```
(set-macro-character #\$
                     #'(lambda (stream char)
                         (declare (ignore char))
                         (round (* 100 (read stream)))))
```

> The rounding step ensures that the amount is a whole number. Binary floating point numbers can not precisely represent all decimal fractions. For example, `(* 100 9.95)` yields `994.9999999999999` and `(* 100 1.10)` yields `110.00000000000001` on my Lisp system.

This says to set `$` to be a macro character which, when encountered by the reader, calls `READ` to get a number and return the nearest whole number after multiplying by 100. It's used like this:

```
? $9.95
995

? $-7.10
-710
```

Now that you can enter dollar amounts directly, you may want to modify the check-writing application to print amounts in whole cents as dollars and cents. To do this, you would redefine the `CHECK` structure with a custom print function, as follows:

```
(defstruct (check
            (:print-function
             (lambda (check stream depth)
               (declare (ignore depth))
               (format stream "#S(CHECK NUMBER ~S DATE ~S AMOUNT $~,2,-2F PAYEE ~S MEMO ~S)"
                       (check-number check)
                       (check-date check)
                       (check-amount check)
                       (check-payee check)
                       (check-memo check)))))
  number date amount payee memo)
```

Then, the `$` reader macro and the `CHECK` print function for its `AMOUNT` slot complement each other perfectly:

```
? (make-check :amount $9.95)
#S(CHECK NUMBER NIL DATE NIL AMOUNT $9.95 PAYEE NIL MEMO NIL)
```

Contents | Cover
Chapter 3 | Chapter 4 | Chapter 5

Copyright © 1995-2001, David B. Lamkins
All Rights Reserved Worldwide

This book may not be reproduced without the written consent of its author. Online distribution is restricted to the author's site.

Chapter 5 - Introducing Iteration

Lisp has several ways to do iteration. In this section we'll look at the most common looping constructs. Later, in Chapter 12, we'll look at mapping, then we'll take a brief look at *series* in Chapter 32; both of these are closely related to iteration.

Simple LOOP loops forever...

The simplest loop in Lisp is just a `LOOP` form wrapped around whatever you want to repeat. Before you try this next bit of code, know how to interrupt execution of your Lisp system; normally this is Command-period on a Macintosh or Control-Break on a PC.

```
? (loop
    (print "Look, I'm looping!"))
"Look, I'm looping!"
"Look, I'm looping!"
"Look, I'm looping!"
"Look, I'm looping!"
"Look, I'm looping!"
"Look, I'm looping!"
"Look, I'm looping!"
"Look, I'm looping!"
... and so on, until you interrupt execution...
Aborted
?
```

This kind of endless loop has legitimate applications. You're already familiar with one: `(LOOP (PRINT (EVAL (READ))))`, Lisp's read-eval-print loop.

> Actually, your Lisp system does some extra things in its read-eval-print loop:
>
> - it catches all errors to prevent you from inadvertently breaking out of the loop
> - it provides a controlled way to exit the loop
> - it keeps track of the most recently entered expressions, results, and printed output

But there's a way out!

Most of the time you write a `LOOP` form, you'd like to have a way out. Fortunately, a `RETURN` form anywhere inside will cause control to leave the `LOOP`; any value you specify becomes the value of the `LOOP` form:

```
? (loop
    (print "Here I am.")
    (return 17)
    (print "I never got here."))
"Here I am."
17
```

`RETURN` is normally used in a conditional form, like this:

```
? (let ((n 0))
    (loop
      (when (> n 10) (return))
      (print n) (prin1 (* n n))
      (incf n)))
0 0
1 1
2 4
3 9
4 16
5 25
6 36
7 49
8 64
9 81
10 100
NIL
?
```

This example could be done better using a `DOTIMES` form, see below. But the combination of `LOOP` and `RETURN` offers the flexibility to return from the middle of a loop, or even from several places within the loop if need be.

Use DOTIMES for a counted loop

To simply loop for some fixed number of iterations, the `DOTIMES` form is your best choice. The previous example simplifies to:

```
? (dotimes (n 11)
    (print n) (prin1 (* n n)))
0 0
1 1
2 4
3 9
4 16
5 25
6 36
7 49
8 64
9 81
10 100
NIL
?
```

`DOTIMES` always returns NIL (or the result of evaluating its optional third argument).

Use DOLIST to process elements of a list

Another common use for iteration is to process each element of a list. `DOLIST` supports this:

```
? (dolist (item '(1 2 4 5 9 17 25))
    (format t "~&~D is~:[n't~;~] a perfect square.~%" item (integerp (sqrt item))))
1 is a perfect square.
2 isn't a perfect square.
4 is a perfect square.
5 isn't a perfect square.
9 is a perfect square.
17 isn't a perfect square.
25 is a perfect square.
NIL
```

> In this example, we've done some fancy things with `FORMAT`. If you want to learn more about what `FORMAT` can do, you should look ahead now to Chapter 24.

The preceding code used a list of numbers, but Lisp allows a list to contain any kind of object:

```
? (dolist (item `(1 foo "Hello" 79.3 2/3 ,#'abs))
    (format t "~&~S is a ~A~%" item (type-of item)))
1 is a FIXNUM
FOO is a SYMBOL
"Hello" is a (SIMPLE-BASE-STRING 5)
79.3 is a DOUBLE-FLOAT
2/3 is a RATIO
#<Compiled-function ABS #x1E9CC3E> is a FUNCTION
NIL
?
```

> Note how we used the backquote and comma to build the list in this example. Do you understand why we did this? All of the list elements up through the ratio `2/3` are self-evaluating; we could have put them in a quoted list as we did in the previous example. But `#'abs` is equivalent to `(function abs)` which, when quoted, is just a list of two symbols. To get the function itself into the quoted list, we had to force evaluation of the `#'abs` form, thus the comma inside the backquoted list.

Like `DOTIMES`, `DOLIST` always returns `NIL` (or the result of its optional third argument).

DO is tricky, but powerful

The `DO` form lets you iterate over multiple variables at the same time, using arbitrary forms to step each variable to its next value. Here's an example which both iterates over the elements of a list and runs a counter at the same time:

```
? (do ((which 1 (1+ which))
       (list '(foo bar baz qux) (rest list)))
      ((null list) 'done)
    (format t "~&Item ~D is ~S.~%" which (first list)))
Item 1 is FOO.
Item 2 is BAR.
Item 3 is BAZ.
Item 4 is QUX.
DONE
?
```

To understand this better, let's look at the general syntax of `DO`, and relate its parts to the example:

```
(do ((var1 init1 step1)
     (var2 init2 step2)
     ...)
    (end-test result)
  statement1
  ...)
```

```
var1        = which
init1       = 1
step1       = (1+ which)
var2        = list
init2       = '(foo bar baz qux)
```

```
step2      = (rest list)
end-test   = (null list)
result     = 'done
statement1 = (format t "~&Item ~D is ~S.~%" which (first list))
```

Contents | Cover
Chapter 4 | Chapter 5 | Chapter 6

Copyright © 1995-2001, David B. Lamkins
All Rights Reserved Worldwide

This book may not be reproduced without the written consent of its author. Online distribution is restricted to the author's site.

Chapter 6 - Deeper into Structures

Structures were introduced in Chapter 3. In this chapter, we'll look at the most useful optional features of structures.

Default values let you omit some initializers, sometimes

Normally, if you create a new structure without specifying a value for some slot, that slot will default to `NIL`.

```
? (defstruct foo-struct a b c)
FOO-STRUCT

? (let ((foo-1 (make-foo-struct :a 1 :b "two")))
    (print (foo-struct-b foo-1))
    (print (foo-struct-c foo-1))
    (values))
"two"
NIL
```

> NOTE: We use the `(values)` form to suppress the return value from the `LET` form. Otherwise, we would have seen one more `NIL` printed.

In cases where `NIL` is a reasonable default value, this behavior is acceptable. But if the normal value of a slot is numeric, for example, you'd really like to start with a reasonable default value rather than having to add a test in all of the code which uses a structure. The full form of a slot specification is a list of the slot name, its default value, and additional options; specifying a bare name instead of the complete list is shorthand for "default value of NIL, and no options."

```
? (defstruct ship
    (name "unnamed")
    player
    (x-pos 0.0)
    (y-pos 0.0)
    (x-vel 0.0)
    (y-vel 0.0))
SHIP
```

When we instantiate this structure using `(MAKE-SHIP)`, the `NAME` slot defaults to `"unnamed"`, the `PLAYER` slot defaults to `NIL`, and the position and velocity slots all default to `0.0`.

Of course, we can still specify slot values to override the defaults:

```
? (make-ship :name "Excalibur" :player "Dave" :x-pos 100.0 :y-pos 221.0)
#S(SHIP :NAME "Excalibur" :PLAYER "Dave" :X-POS 100.0 :Y-POS 221.0 :X-VEL 0.0 :Y-VEL 0.0)
```

Lisp's default printer for structures makes it easy to see the slots and their values. We've given explicit values to all of the slots except the two velocity slots, which have their default values.

Change the way Lisp prints your structures

To print a structure using other than the default printer, you may define a new print function as a structure option.

```
? (defstruct (ship
              (:print-function
               (lambda (struct stream depth)
                 (declare (ignore depth))
                 (format stream "[ship ~A of ~A at (~D, ~D) moving (~D, ~D)]"
                         (ship-name struct)
                         (ship-player struct)
                         (ship-x-pos struct)
                         (ship-y-pos struct)
                         (ship-x-vel struct)
                         (ship-y-vel struct)))))
    (name "unnamed")
    player
    (x-pos 0.0)
    (y-pos 0.0)
    (x-vel 0.0)
    (y-vel 0.0))
SHIP

? (make-ship :name "Proud Mary" :player 'CCR)
[ship Proud Mary of CCR at (0.0, 0.0) moving (0.0, 0.0)]
```

Actually, it's considered bad practice to print something the reader can't interpret. Our use of the brackets around the printed ship description is not necessarily good or bad, it depends upon how the current read table is specified (we first saw reader macros in Chapter 3, Lesson 12.

One way to ensure that the reader doesn't get confused is to deliberately print something so as to be unreadable. By convention, Lisp prints such objects beginning with `#<`. You could change your format string to read `"#<ship ~A of ~A at (~D, ~D) moving (~D, ~D)>"`, so the prior `MAKE-SHIP` example would print `#<ship Proud Mary of CCR at (0.0, 0.0) moving (0.0, 0.0)>`. However, since 1990 Lisp systems have had a `PRINT-UNREADABLE-OBJECT` macro which should be used for this purpose. If the printer control variable `*PRINT-READABLY*` is true, `PRINT-UNREADABLE-OBJECT` will signal an error.

```
;; Use PRINT-UNREADABLE-OBJECT macro -- changes in boldface
? (defstruct (ship
              (:print-function
               (lambda (struct stream depth)
                 (declare (ignore depth))
                 (print-unreadable-object (struct stream)
                   (format stream "ship ~A of ~A at (~D, ~D) moving (~D, ~D)"
                           (ship-name struct)
                           (ship-player struct)
                           (ship-x-pos struct)
                           (ship-y-pos struct)
                           (ship-x-vel struct)
                           (ship-y-vel struct))))))
    (name "unnamed")
    player
    (x-pos 0.0)
    (y-pos 0.0)
    (x-vel 0.0)
    (y-vel 0.0))
SHIP
```

Alter the way structures are stored in memory

Lisp stores structures in an implementation-dependent manner unless you specify otherwise using a structure option. You have two choices if you decide to specify structure storage: store it as a vector (possibly with a particular type for all of the elements) or as a list. Here, we use the untyped vector option -- the list option is similar:

```
? (defstruct (bar
              (:type vector))
    a b c)
BAR

? (make-bar)
#(NIL NIL NIL)
```

Note that the slot names are not stored when you specify the storage type. This is probably the biggest advantage for using this option -- it can save storage in the amount of a machine word per slot per instance. The disadvantage is that Lisp does not recognize such a structure as a distinct type, and does not create a `<structure-name>-P` predicate for you.

If you are satisfied with being able to retrieve the name of the structure, but still want the storage savings associated with specifying the structure's representation, you can do this:

```
? (defstruct (bar
              (:type vector)
              :named)
    a b c)
BAR

? (make-bar)
#(BAR NIL NIL NIL)
```

Using the list representation option has the drawbacks noted above, but none of the advantages; the backbone of the list typically *adds* a machine word of storage per slot when compared to the default representation, which is usually a vector. The only time it would make sense to explicitly specify a list representation is when the default structure representation is list-based or when the Lisp implementation imposes some artificial limit on the space reserved for storage of vectors; neither case applies in modern implementations.

Shorten slot accessor names

Slot accessor names are constructed from the name of the structure and the slot. If the structure and the slot both have lengthy names, the accessor names can get unwieldy. You can abbreviate names somewhat by using the `:CONC-NAME` structure option to specify a name to use instead of the structure name.

```
? (defstruct (galaxy-class-cruiser-ship
              (:conc-name gcc-ship-)) ; name includes trailing hyphen!
    name player (x-pos 0.0) (y-pos 0.0) (x-vel 0.0) (y-vel 0.0))
GALAXY-CLASS-CRUISER-SHIP

? (let ((ship (make-galaxy-class-cruiser-ship)))
    (print (gcc-ship-x-pos ship)) ; note abbreviated accessor name
    (values))
0.0
```

Allocate new structures without using keyword arguments

For certain structures, it may be more convenient to make a new instance using just a list of arguments instead of keywords and arguments. You can redefine a structure constructor's argument list using the `:CONSTRUCTOR` option.

```
? (defstruct (3d-point
              (:constructor
               create-3d-point (x y z)))
    x y z)
3D-POINT

? (create-3d-point 1 -2 3)
#S(3D-POINT :X 1 :Y -2 :Z 3)
```

> NOTE: The slot values do *not* default to `NIL` if you use a `:CONSTRUCTOR` option!

Most lambda-list options are available to the constructor function -- consult a Lisp reference manual for details.

Define one structure as an extension of another

We use inheritance to define one object in terms of another. Structures permit a very simple form of inheritance using the `:INCLUDE` option.

```
? (defstruct employee
    name department salary social-security-number telephone)
EMPLOYEE

? (make-employee)
#S(EMPLOYEE :NAME NIL :DEPARTMENT NIL :SALARY NIL :SOCIAL-SECURITY-NUMBER NIL :TELEPHONE NIL)

? (defstruct (manager
              (:include employee))
    bonus direct-reports)
MANAGER

? (make-manager)
#S(MANAGER :NAME NIL :DEPARTMENT NIL :SALARY NIL :SOCIAL-SECURITY-NUMBER NIL :TELEPHONE NIL :BONUS NI
```

All accessors which apply to an `EMPLOYEE` also apply to a `MANAGER`, and a `MANAGER` instance is also an `EMPLOYEE` instance. Notice in the following example how the `...-NAME` accessors for both `MANAGER` and `EMPLOYEE` reference the same slot.

```
? (setq mgr (make-manager))
#S(MANAGER :NAME NIL :DEPARTMENT NIL :SALARY NIL :SOCIAL-SECURITY-NUMBER NIL :TELEPHONE NIL :BONUS NI

? (setf (manager-name mgr) "Buzz")
"Buzz"

? (employee-name mgr)
"Buzz"
```

A structure may have one `:INCLUDE` option, at most. This limits the ability of structures to model the real world by describing inheritance. CLOS objects allow multiple inheritance, and have many other useful and convenient features. We will get our first look at CLOS in Chapter 7.

Contents | Cover
Chapter 5 | Chapter 6 | Chapter 7

Copyright © 1995-2001, David B. Lamkins
All Rights Reserved Worldwide

This book may not be reproduced without the written consent of its author. Online distribution is restricted to the author's site.

Chapter 7 - A First Look at Objects as Fancy Structures

We first encountered structures in Chapter 3, then learned about some of their optional behavior in Chapter 6. In this chapter we'll start to learn about objects in the Common Lisp Object System (CLOS). For now, we'll look at just the ways objects can be used to structure data. Later, in Chapter 14, we'll learn more about CLOS.

Hierarchies: Classification vs. containment

When you program with objects you will attempt, in some way, to create a model of some portion of the real world. When you do this, you'll probably notice that some objects are made up of smaller parts. Each part has its own identity; the part is identifiable by itself, separate from any object that it may be a part of. Furthermore, a part may be made from smaller parts. If you drew a picture of the component relationships among all the parts of some complex object, you'd find that they formed a hierarchy. The fully-assembled object will be at the top of the hierarchy (the first level), all of its pieces will be at the second level, all of the pieces that make up the second-level parts will be at the third level, and so on. This hierarchy is a *containment hierarchy*; each level represents an object, and the next lowest level represents the objects that are parts of the object at the higher level. An object at a higher level contains (or has as parts) some objects at a lower level in the hierarchy, and an object at a lower level is contained by (or is part of) some object at a higher level.

Containment hierarchies are important because they model "has-a" and "is-a-part-of" relationships among objects. These relationships simplify your program's model of the real world by letting you think in terms of relatively small component parts, rather than having to model a single, highly complex object. Modeling by containment also pays off when you can model a similar object in terms of a different combination of components.

> An object may have certain characteristics which can not be separated from the object. For example, an object may have color, size, mass, velocity, and temperature. These characteristics are *not* component parts of the object; they can not be separated from the object, nor can they be combined to create new objects. These characteristics are *attributes* of the object.

The other kind of hierarchy you'll work with as an object programmer is a *classification hierarchy*. In a classification hierarchy, objects are connected by "is-a-kind-of" (or more concisely, "is-a" or "a-k-o") relationships. These relationships also have different names depending upon our point of view: if A is a kind of B then A is a specialization of B, while B is a generalization of A.

> With the explosion of interest in object programming, many specialized lexicons have grown up to support specific methods and languages. After you eliminate the terms that describe special features of a particular methodology or language implementation, what's left is usually a renaming of containment and classification hierarchies (and the relationships supported by each) and some way to specify object attributes. In fact, the most common renaming is to refer to a classification hierarchy as a "class" hierarchy.

Use DEFCLASS to define new objects

A CLOS object is defined by a `DEFCLASS` form. `DEFCLASS` only *describes* an object. To create an instance of an object, you can use a `MAKE-INSTANCE` form.

Here's how you would define a trivial object:

```
? (defclass empty-object () ())
#<STANDARD-CLASS EMPTY-OBJECT>
```

This class is not very interesting; the `()` are placeholders for things to come. Also, note that this particular Lisp system prints `#<STANDARD-CLASS EMPTY-OBJECT>` in response to the `DEFCLASS` form. This is unreadable -- the reader signals an error whenever it reads a form that begins with `#<` -- but it lets you know that something useful happened.

Once you've defined a class, you can use it to make objects. Most forms that require a class will accept the name of the class, or you can use `FIND-CLASS` to retrieve the actual class, given its name. `MAKE-INSTANCE` creates a new object, given a class name or a class:

```
? (make-instance 'empty-object)
#<EMPTY-OBJECT #x3CA1206>
? (make-instance 'empty-object)
#<EMPTY-OBJECT #x3CA1DFE>
? (find-class 'empty-object)
#<STANDARD-CLASS EMPTY-OBJECT>
? (make-instance (find-class 'empty-object))
#<EMPTY-OBJECT #x3CB397E>
```

Again, the Lisp system responds with an unreadable object. This time, the response includes the storage address of the object. Most Lisp systems have a default printer for CLOS objects that works like this, even though the details may differ slightly. The important thing to note is that `MAKE-INSTANCE` creates a new object each time it is called.

> Even though the object printer shows a different address for each object, you should *not* depend upon the printed representation to identify an object. Lisp systems can (and do) change the address of objects at runtime. The only way to reliably compare the identity of an object is with an identity test such as `EQ` (see Chapter 17).

Objects have slots, with more options than structures

Our first `DEFCLASS` form defined an object that wasn't good for much of anything. Now we'll see how to expand a class definition so that our objects will have named slots. These serve the same purpose as slots in structures (Chapter 3), they simply store data. The data could be attributes of the object, or contained objects, or references to related objects.

To define a class for an object with slots, we start with a `DEFCLASS` form and add slot definitions, like this:

```
? (defclass 3d-point () (x y x))
#<STANDARD-CLASS 3D-POINT>
```

Here, we've defined a class `3D-POINT` whose objects will have three slots, named `X`, `Y`, and `Z`. This looks like it might be similar to a structure definition, such as

```
(defstruct 3d-point-struct x y z)
```

but the class actually has **less** functionality than the structure. The class does not define default accessors for slots. To access the slots, you would have to use `SLOT-VALUE` as in this example:

```
? (let ((a-point (make-instance '3d-point)))
    (setf (slot-value a-point 'x) 0)  ; set the X slot
    (slot-value a-point 'x))          ; get the X slot
0
```

If you try to get the value of a slot before setting it, Lisp will signal an error because the slot is unbound (i.e. it has *no* value).

```
? (let ((a-point (make-instance '3d-point)))
    (slot-value a-point 'y))
> Error: Slot Y is unbound in #<3D-POINT #x3CD3216>
```

Controlling access to a slot helps keep clients honest

Getting and setting slots with `SLOT-VALUE` forms is slightly cumbersome when compared to the accessors created automatically for slots in a structure. Fortunately, you can specify accessors for each slot when you define a class.

```
(defclass 3d-point ()
  ((x :accessor point-x)
   (y :accessor point-y)
   (z :accessor point-z)))
```

Then, object slot access looks just like structure slot access.

```
? (let ((a-point (make-instance '3d-point)))
    (setf (point-x a-point) 0)
    (point-x a-point))
0
```

You can also specify separate accessor names for reading and writing a slot.

```
? (defclass 3d-point ()
    ((x :reader get-x :writer set-x)
     (y :reader get-y :writer set-y)
     (z :reader get-z :writer set-z)))
#<STANDARD-CLASS 3D-POINT>
? (let ((a-point (make-instance '3d-point)))
    (set-z 3 a-point)
    (get-z a-point))
3
```

Do you see the difference between specifying `:accessor` and `:writer`? Notice that the slot writer is *not* used with `SETF`. Also note the order of arguments to the slot writer: first the value, then the object.

It's useful to have all of these options for slot access when you're writing a complex program. Through various combinations of slot accessor definitions, you can give a slot read/write, read-only, write-only, or no access. You might think that the last two cases wouldn't be useful, but they are. A write-only slot might provide information that is only useful to establish a state within the object in response to a request from the object's client -- a seed for a random number generator, for example. A no-access slot can maintain information that should be known only by the internal workings of the object; functions that manipulate the object's internal state can still access the slot using `SLOT-VALUE`.

The existence of `SLOT-VALUE` is anathema to some object designers, who believe that the privacy of an object's internal information should be *absolutely* protected against client access. Lisp requires the exercise of programmer discipline to protect an object's private information. As a rule of thumb, you should use `SLOT-VALUE` to manipulate private slots, and provide appropriate named accessors for all other slots. Having done so, any appearance of a `SLOT-VALUE` form in a client program signals a violation of your intent to hide some object's private state.

Override a slot accessor to do things that the client can't

In this section we'll see how to define special-purpose accessors that do more than just read and write slot values. Think of this as a sneak preview of Chapter 14.

Let's define a sphere. A sphere is defined by its position in 3-space and by its radius. We'd like to query the sphere for its volume. Finally, we'd like to be able to move -- or translate -- the sphere's position by a specified amount without having to explicitly calculate its new coordinates. We start with this class definition:

```
? (defclass sphere ()
    ((x :accessor x)
     (y :accessor y)
     (z :accessor z)
     (radius :accessor radius)
     (volume :reader volume)
     (translate :writer translate)))
#<STANDARD-CLASS SPHERE>
```

The accessors for `X`, `Y`, `Z`, and `RADIUS` need no further explanation, but the accessors for `VOLUME` and `TRANSLATE` aren't yet useful; the `VOLUME` reader will fail because its slot is unbound and the `TRANSLATE` writer won't do anything except to set its slot.

We'll finish the definition of our sphere by first having `VOLUME` return a value calculated from the sphere's radius. There are two ways to do this: have `VOLUME` read the sphere's radius and calculate the corresponding volume, or have `RADIUS` calculate the volume and set the volume slot for later use by the `VOLUME` accessor. Here are both solutions -- if you try this out, pick just one:

```
; Volume from Radius
(defmethod volume ((object sphere))
  (* 4/3 pi (expt (radius object) 3)))

; Radius to Volume
(defmethod radius ((new-radius number) (object sphere))
  (setf (slot-value object 'radius) new-radius)
  (setf (slot-value object 'volume)
        (* 4/3 pi (expt new-radius 3))))
```

> This is not the best example of Lisp style. You're defining a default reader or writer method in the `DEFCLASS` form, then redefining the method to add special behavior. Your Lisp system may warn you about the attempted redefinition; it's OK to continue from the warning and redefine the method.
>
> One way to avoid the problem is to omit the slot option that defines the default `:READER` (for `VOLUME`) or `:WRITER` (for `TRANSLATE`) in the `DEFCLASS` form, but then you lose the documentation provided by these declarations. We'll see some other declarations in Chapter 14 that can help to improve readability.

The `DEFMETHOD` form defines a function which applies only to certain argument types. In this case, the `VOLUME` method applies only to `SPHERE` objects, and the `RADIUS` method applies only to a `NUMBER` (the new radius) and a `SPHERE`. The `VOLUME` method computes the volume from the sphere's radius each time it is called. The `RADIUS` method computes the sphere's volume each time the radius is set; the values of both radius and volume are stored in the sphere's slots by the `SLOT-VALUE` forms.

You can think of a default slot reader and writer as being defined like this (but the compiler probably generates better code if you just use the default accessors created automatically by the `:READER` and `:WRITER` slot options):

```
; Default slot reader (illustration only)
(defmethod slot-reader ((object object-class))
  (slot-value object 'slot-name))

; Default slot writer (illustration only)
(defmethod slot-writer (new-value (object object-class))
  (setf (slot-value object 'slot-name) new-value))
```

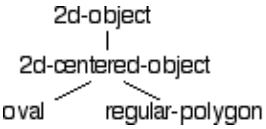
Define classes with single inheritance for specialization

Specialization is one of the most important concepts in object programming. Specialization allows you to define an object in terms of another object by describing features that are new or different; the base functionality of the object is *inherited* from the definition of the parent.

In the simplest kind of specialization, single inheritance, a child inherits traits from just one parent. As an example, we'll define some basic two dimensional objects using a single inheritance hierarchy.

```
(defclass 2d-object () ())

(defclass 2d-centered-object (2d-object)
  (x :accessor x)
  (y :accessor y)
  (orientation :accessor orientation)

(defclass oval (2d-centered-object)
  (axis-1 :accessor axis-1)
  (axis-2 :accessor axis-2))

(defclass regular-polygon (2d-centered-object)
  (n-sides :accessor number-of-sides)
  (size :accessor size))
```

The inheritance graph for these four classes looks like this:



The `2D-OBJECT` class is a placeholder from which we might later derive other 2D objects, e.g. lines and points. A `2D-CENTERED-OBJECT` has a central reference position specified by its `X` and `Y` attributes, and an orientation -- the amount the object is rotated about its central position. The `OVAL` and `REGULAR-POLYGON` classes inherit from `2D-CENTERED-OBJECT`, retaining the position and orientation attributes of the parent class and adding attributes appropriate to the geometry of the derived objects.

Multiple inheritance allows mix-and-match definition

CLOS supports multiple inheritance, which lets a class inherit traits from more than one parent. This is useful for a programming style that starts with common functionality and then "mixes in" extensions to the basic behavior.

As an example, let's suppose that we'd like to write code to render the 2D objects we started to define in the previous section. Let's say that we'd like to achieve two goals simultaneously: we'd like to render images on either a bitmapped or a Postscript display device, and we'd like to render the objects either on a plane surface or as a projection in a "2.5D" space, where each 2D object has a Z depth, and the system provides a choice of view positions. These requirements are not at all interdependent; one deals with the details of rendering objects on a display device, while the other deals with transformations of the objects that must occur prior to rendering.

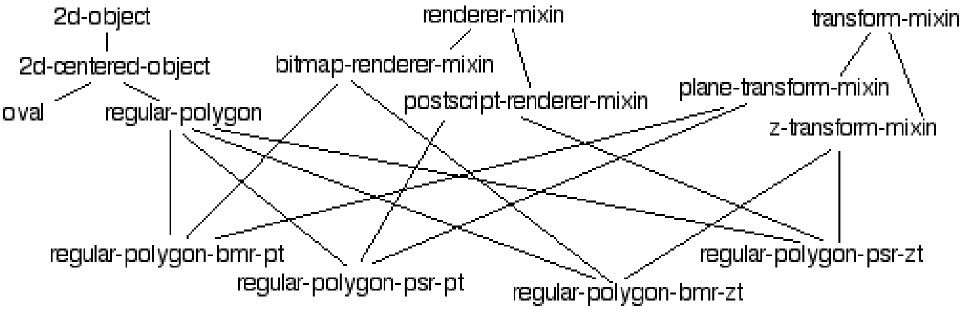
One way to address these requirements is by using multiple inheritance. One set of of mixins handles transforms, while another set of mixins handles the details of rendering to a display device. With a carefully designed protocol for sharing information, new combinations of transforms and renderers can be added to our 2D objects without rewriting any existing code.

We saw in the previous section how single inheritance is written using `DEFCLASS`, by putting the parent's class name within the first parentheses following the new class name. Classes that inherit from multiple parents simply list all of the parents. Thus to define all the combinations of transform and rendering for our `REGULAR-POLYGON` class, we could do something like this:

```
(defclass renderer-mixin () (...))
(defclass bitmap-renderer-mixin (renderer-mixin) (...))
(defmethod render (description (self bitmap-renderer-mixin)) ...)
(defclass postscript-renderer-mixin (renderer-mixin) (...))
(defmethod render (description (self postscript-renderer-mixin)) ...)

(defclass transform-mixin () (...))
(defclass plane-transform-mixin (transform-mixin) (...))
(defmethod transform ((self plane-transform-mixin)) ...)
(defclass z-transform-mixin (transform-mixin) (...))
(defmethod transform ((self z-transform-mixin)) ...)

(defmethod draw ((self regular-polygon))
  (render (transform self) self))

(defclass regular-polygon-bmr-pt (regular-polygon
                                  bitmap-renderer-mixin
                                  plane-transform-mixin)
  (...))

(defclass regular-polygon-psr-pt (regular-polygon
                                  postscript-renderer-mixin
                                  plane-transform-mixin)
  (...))

(defclass regular-polygon-bmr-zt (regular-polygon
                                  bitmap-renderer-mixin
                                  z-transform-mixin)
  (...))

(defclass regular-polygon-psr-zt (regular-polygon
                                  postscript-renderer-mixin
```

```
                                    z-transform-mixin)
  (...))
```

Now our class hierarchy looks like this:


2d-object
renderer-mixin
transform-mixin
2d-centered-object
bitmap-renderer-mixin
oval
regular-polygon
postscript-renderer-mixin
plane-transform-mixin
z-transform-mixin
regular-polygon-bmr-pt
regular-polygon-psr-pt
regular-polygon-bmr-zt
regular-polygon-psr-zt

I've shown a few method definitions to give you an idea for how the drawing protocol interacts with the mixin definitions to generate the expected behavior. The `DRAW` method specialized on the `REGULAR-POLYGON` class provides the protocol: it calls `TRANSFORM` to apply a transform to itself, then `RENDER` to draw itself, using some (as yet unspecified) description generated by `TRANSFORM`.

Now let's assume that we've created an instance of `REGULAR-POLYGON-BMR-PT` and have called the `DRAW` method:

```
(let ((poly (make-instance 'regular-polygon-bmr-pt ...)))
  (draw poly))
```

The `DRAW` method is *not* specialized on the `REGULAR-POLYGON-BMR-PT` class, so we invoke the more general method specialized on `REGULAR-POLYGON`. This `DRAW` method then attempts to invoke methods `TRANSFORM` and `RENDER` which are specialized on the `REGULAR-POLYGON-BMR-PT` class; these methods are defined, so they provide the mixin behavior we'd expect.

As you can see from this very simple example, mixins are a powerful tool for object programming. Having predefined a very simple protocol -- the `DRAW` method which invokes `TRANSFORM` and `RENDER` -- we can add new behaviors to our system by defining additional mixins. The original code is unchanged. Because Lisp can add definitions dynamically, you don't even have to stop your software to extend it in this manner.

Options control initialization and provide documentation

By default, an object's slots are unbound in a new object. In many cases it would be more useful to give slots some meaningful initial value. For example, our `3D-POINT` could be initialized to the origin.

```
(defclass 3d-point ()
  ((x :accessor point-x :initform 0)
   (y :accessor point-y :initform 0)
   (z :accessor point-z :initform 0)))
```

The `:INITFORM` slot option provides a value for the slot at the time the object is created. The initialization form is evaluated every time it is used to initialize a slot.

You might also want to provide specific initialization arguments when an object is created. To do this, use the `:INITARG` slot option.

```
(defclass 3d-point ()
  ((x :accessor point-x :initform 0 :initarg :x)
   (y :accessor point-y :initform 0 :initarg :y)
   (z :accessor point-z :initform 0 :initarg :z)))
```

To create a `3D-POINT` object using explicit initializers, you'd do something like this:

```
(make-instance '3d-point :x 32 :y 17 :z -5)
```

Because the class definition includes both `:INITFORM` and `:INITARG` options, the slot will still get its default value if you omit an explicit initializer.

Slot definitions also allow `:DOCUMENTATION` and `:TYPE` options.

```
(defclass 3d-point ()
  ((x :accessor point-x :initform 0 :initarg :x
      :documentation "x coordinate" :type real)
   (y :accessor point-y :initform 0 :initarg :y
      :documentation "y coordinate" :type real)
   (z :accessor point-z :initform 0 :initarg :z
      :documentation "z coordinate" :type real)))
```

The `:TYPE` option may be used by the compiler to assist in code optimization or to create runtime tests when setting slots. However, this behavior may vary among Lisp compilers, and the standard does not require `any` interpretation of the option. Therefore, it is best to think of this as additional documentation to the person reading your program.

This is only the beginning...

In Chapter 14 we'll examine methods in greater depth and see how to associate behaviors with objects.

Contents | Cover
Chapter 6 | Chapter 7 | Chapter 8

Copyright © 1995-2001, David B. Lamkins
All Rights Reserved Worldwide

This book may not be reproduced without the written consent of its author. Online distribution is restricted to the author's site.

Chapter 8 - Lifetime and Visibility

In this chapter we'll see how lifetime and visibility affect the values of Lisp variables during execution. This is pretty much like local and global variables in other languages, but Lisp's special variables change things. This chapter also sets the stage for understanding that lifetime and visibility aren't just for variables.

Everything in Lisp has both lifetime and visibility

Every object in Lisp has both lifetime and visibility. We'll see why this is important in the following sections.

Lifetime: Creation, existence, then destruction

An object's lifetime is the period between its creation and destruction. Some objects have fleeting lifetimes, limited to the form in which they appear. Other objects are created as soon as the program begins running, and are not destroyed until the program finishes. And others enter and leave existence according to still other rules.

Visibility: To see and to be seen by

An object is either visible or not visible at a particular point in your program. Sometimes visibility is controlled by the execution path of the program. But for most objects in Common Lisp, visibility is determined by the textual arrangement of your program; this is good, because you can reason about visibility just by reading a program, without having to first reason about the program's control flow.

The technical names: Extent and Scope

When you read language specifications for Common Lisp, you'll see the technical terms extent and scope used in place of lifetime and visibility. I wanted to introduce these concepts first using the non-technical terms because I believe them to be more evocative. But you should get used to reading about extent and scope. Remember:

> **lifetime** is to **extent**
> as
> **visibility** is to **scope**

Really easy cases: top-level defining forms

Top-level defining forms are easy. The objects so defined have *indefinite* extent and scope. This is a fancy way of saying that objects defined by top level forms "always" exist and are visible (or at least potentially accessible, as I'll explain shortly) everywhere in the program.

Practically, what this means is that every object defined by a top level form exists for as long as the program runs. The object comes into existence when the top level form is evaluated. If the form was compiled into a file, then the object created by the form comes into existence when the compiled file is loaded.

An object having indefinite scope is visible everywhere in your program. It doesn't matter whether the object was created in a different source file or at a different time -- even if it was created *after* you define a function that references the top level object (although some Lisp compilers will issue a warning when you do this, the code will always behave properly if you evaluate the object's defining form before evaluating the function that references the object).

> If you're familiar with the concept of lexical scope as it applies to programming languages, you're probably confused by the notion of indefinite scope. If I introduce an object whose name *shadows* the name of an object in an outer scope, then that outer object is "not visible" in the inner scope. And you're right, up to a point.
>
> Lisp makes a very clear and explicit distinction between an object and its name. We say that an object is bound to a name, or that a (named) binding is established for an object. And it is very true that a *binding* in an inner lexical scope may shadow a *binding* in an outer scope. However, the scope of the outer object extends into the inner scope, even though it is inaccessible via its shadowed binding. This is an important distinction, because an object may have more than one binding, and the object must remain accessible via any binding which has not been lexically shadowed.

Scope and extent of parameters and LET variables

Objects bound by function parameters and `LET` forms have lexical scope. Their bindings are visible beginning at a certain textual location in the defining form and continuing through the textual end of the defining form. Any reference to a textually identical name from outside of the defining form must refer to a different binding. A nested defining form may declare a binding that shadows an enclosing binding to a textually identical name.

> This is a slightly more rigorous restatement of concepts introduced in Chapter 3, Lesson 6. If you need to refresh your memory, this would be a good time to go back and review the examples in that short passage.

Slightly trickier: special variables

Special variables (also known by the more technically correct term *dynamic variables*) have *dynamic* scope. This means that a binding is created for a special variable as a result of executing some form in your program. The scope of the dynamic binding extends into any form called (directly or indirectly) by the form which established the dynamic binding.

The extent of a special variable lasts indefinitely, until the form that created the dynamic binding is no longer a site of active program execution -- in other words, until the defining form (and all of the forms called by it) finishes executing. If the dynamic binding is created by a top level form, the extent is the same as described previously for top level defining forms.

```
? (defparameter *my-special-variable* 17)
*MY-SPECIAL-VARIABLE*
? (defun show-my-special ()
    (declare (special *my-special-variable*))
    (print *my-special-variable*)
    nil)
SHOW-MY-SPECIAL
? (defun do-something-else ()
    (show-my-special))
DO-SOMETHING-ELSE
? (defun dynamically-shadow-my-special ()
    (let ((*my-special-variable* 8))
      (do-something-else))
    (show-my-special))
DYNAMICALLY-SHADOW-MY-SPECIAL
? (dynamically-shadow-my-special)

8
17
NIL
```

When reading the above, pay special attention to `DO-SOMETHING-ELSE` -- this calls `SHOW-MY-SPECIAL`. `SHOW-MY-SPECIAL` would normally see the lexical value of `*MY-SPECIAL-VARIABLE*` -- 17 -- except for the declaration which says that `*MY-SPECIAL-VARIABLE*` is a special variable.

`DYNAMICALLY-SHADOW-MY-SPECIAL` binds `*MY-SPECIAL-VARIABLE*` to the value 8, then calls `DO-SOMETHING-ELSE`, which in turn calls `SHOW-MY-SPECIAL`. At this point, the `LET` binding of `*MY-SPECIAL-VARIABLE*` is not lexically apparent to the code in `SHOW-MY-SPECIAL`. Yet, because the binding is declared special at the point of reference, and because the binding `LET` form is still active when `DO-SOMETHING-ELSE` calls `SHOW-MY-SPECIAL`, the *dynamic* binding of 8 (rather than the lexical binding of 17) is printed.

Later during execution, the second call to `SHOW-MY-SPECIAL` happens outside of the `LET` form, and the top level value of `*MY-SPECIAL-VARIABLE*` -- 17 -- is printed.

> Strictly speaking, the `(DECLARE (SPECIAL ...` form is not necessary in `SHOW-MY-SPECIAL` -- the `DEFPARAMETER` form has the side effect of proclaiming its variable to be special. However, the added declaration adds redundant documentation at the point of use of the special variable. Furthermore, some Lisp compilers will issue a warning (typically: "`Undeclared free variable assumed special`") that is easily silenced by adding the declaration.

Contents | Cover
Chapter 7 | Chapter 8 | Chapter 9

Copyright © 1995-2001, David B. Lamkins
All Rights Reserved Worldwide

This book may not be reproduced without the written consent of its author. Online distribution is restricted to the author's site.

Chapter 9 - Introducing Error Handling and Non-Local Exits

In this chapter we'll see some of the specialized control flow forms provided by Common Lisp.

UNWIND-PROTECT: When it absolutely, positively has to run

One of the challenges of writing robust programs is to make sure that important parts of your code always run, even in the presence of errors. Usually, this is most important when you're allocating, using and releasing resources such as files and memory, like this:

```
; Setup
Allocate some resources
Open some files
; Process
Process using files and storage (may fail)
; Cleanup
Close the files
Release the resources
```

If the processing step might fail (or be interrupted by the user) you should make sure that every possible exit path still goes through the cleanup section to close the files and release the storage. Better still, your program should be prepared to handle errors that occur during the setup phase as you allocate storage and open files, since any of these operations might also fail; any partially completed setup should still be undone in the cleanup section.

Lisp's `UNWIND-PROTECT` form makes this especially easy to do.

```
(let (resource stream)
  (unwind-protect
    (progn
      (setq resource (allocate-resource)
            stream (open-file))
      (process stream resource))
    (when stream (close stream))
    (when resource (deallocate resource))))
```

Here's what happens. The `LET` binds `RESOURCE` and `STREAM` to `NIL` -- we'll use the `NIL` value to mean that there has been no resource allocated or file opened. The first form in the `UNWIND-PROTECT` is a "protected" form; if control leaves the protected form via *any* means, then the rest of the forms -- the "cleanup" forms -- are guaranteed to be executed.

In our example, the protected form is a `PROGN` that calls `ALLOCATE-RESOURCE` and `OPEN-FILE` to set our local variables, then `PROCESS` uses these resources. `SETQ` assigns values sequentially to our local variables: `(ALLOCATE-RESOURCE)` must succeed before a value can be assigned to `RESOURCE`, then `OPEN-FILE` must succeed before its value can be assigned to `STREAM`. A failure (i.e. an interrupt or error) at any point in this sequence will transfer control out of the protected form.

If the initializations succeed and `PROCESS` returns normally, control continues into the cleanup forms.

If anything causes the protected form to exit -- for example, an error or an interrupt from the keyboard --

control is transferred immediately to the first cleanup form. The cleanup forms are guarded by `WHEN` clauses so we won't try to close the stream or deallocate the resource if an error caused them to never be created in the first place.

Gracious exits with BLOCK and RETURN-FROM

The `BLOCK` and `RETURN-FROM` forms give you a structured lexical exit from any nested computation. The `BLOCK` form has a name followed a body composed of zero or more forms. The `RETURN-FROM` form expects a block name and an optional (the default is `NIL`) return value.

```
? (defun block-demo (flag)
    (print 'before-outer)
    (block outer
      (print 'before-inner)
      (print (block inner
               (if flag
                 (return-from outer 7)
                 (return-from inner 3))
               (print 'never-print-this)))
      (print 'after-inner)
      t))
BLOCK-DEMO
? (block-demo t)

BEFORE-OUTER
BEFORE-INNER
7
? (block-demo nil)

BEFORE-OUTER
BEFORE-INNER
3
AFTER-INNER
T
```

When we call `BLOCK-DEMO` with `T`, the `IF` statement's consequent -- `(return-from outer 7)` -- immediately returns the value 7 from the `(BLOCK OUTER ...` form. Calling `BLOCK-DEMO` with `NIL` executes the alternate branch of the `IF` -- `(return-from inner 3)` -- passing the value 3 to the `PRINT` form wrapped around the `(BLOCK INNER ...` form.

Block names have lexical scope: `RETURN-FROM` transfers control to the `innermost` `BLOCK` with a matching name.

Some forms implicitly create a block around their body forms. When a name is associated with the form, such as with `DEFUN`, the block takes the same name.

```
? (defun block-demo-2 (flag)
    (when flag
      (return-from block-demo-2 nil))
    t)
BLOCK-DEMO-2
? (block-demo-2 t)
NIL
? (block-demo-2 nil)
T
```

Other forms, such as the simple `LOOP` and `DOTIMES`, establish a block named `NIL` around their body forms. You can return from a `NIL` block using `(RETURN-FROM NIL ...)`, or just `(RETURN ...)`.

```
? (let ((i 0))
    (loop
      (when (> i 5)
        (return))
      (print i)
      (incf i)))

0
1
2
3
4
5
NIL
? (dotimes (i 10)
    (when (> i 3)
      (return t))
    (print i))

0
1
2
3
T
```

Escape from anywhere (but not at any time) with CATCH and THROW

So `BLOCK` and `RETURN-FROM` are handy for tranferring control out of nested forms, but they're only useful when the exit points (i.e. block names) are lexically visible. But what do you do if you want to break out of a chain of function calls?

```
; WARNING! This won't work!
(defun bad-fn-a ()
  (bad-fn-b))

(defun bad-fn-b ()
  (bad-fn-c))

(defun bad-fn-c ()
  (return-from bad-fn-a))  ; There is no block BAD-FN-A visible here!
```

Enter `CATCH` and `THROW`, which let you establish control transfers using dynamic scope. Recall that dynamic scope follows the chain of active forms, rather than the textual enclosure of one form within another of lexical scope.

```
? (defun fn-a ()
    (catch 'fn-a
      (print 'before-fn-b-call)
      (fn-b)
      (print 'after-fn-b-call)))
FN-A
? (defun fn-b ()
    (print 'before-fn-c-call)
    (fn-c)
    (print 'after-fn-c-call))
FN-B
?(defun fn-c ()
   (print 'before-throw)
   (throw 'fn-a 'done)
   (print 'after-throw))
FN-C
? (fn-a)

BEFORE-FN-B-CALL
BEFORE-FN-C-CALL
BEFORE-THROW
DONE
```

Making sure files only stay open as long as needed

Opening a file just long enough to process its data is a very common operation. We saw above that `UNWIND-PROTECT` can be used to ensure that the file gets properly closed. As you might expect, such a common operation has its own form in Lisp.

```
(with-open-file (stream "file.ext" :direction :input)
  (do-something-with-stream stream))
```

`WITH-OPEN-FILE` wraps an `OPEN` and `CLOSE` form around the code you provide, and makes sure that the `CLOSE` gets called at the right time. All of the options available to `OPEN` may be used in `WITH-OPEN-FILE` -- I've shown the options you'd use to open a file for input.

Contents | Cover
Chapter 8 | Chapter 9 | Chapter 10

Copyright © 1995-2001, David B. Lamkins
All Rights Reserved Worldwide

This book may not be reproduced without the written consent of its author. Online distribution is restricted to the author's site.

Chapter 10 - How to Find Your Way Around, Part 1

In this chapter, you'll learn how to find your way around Common Lisp without resorting so often to the manuals. (You *do* read the fine manuals, don't you?)

Oh, there's one thing you should keep in mind while you're reading this chapter: all of these tools work equally well for the built-in functionality of your Lisp system and for all of the Lisp programs that you write.

APROPOS: I don't remember the name, but I recognize the face

Common Lisp defines 978 symbols. Whatever implementation you use probably defines hundreds of additional symbols for language extensions, additional libraries, a graphical user interface, etc. Are you going to remember the names of all these symbols? Not likely...

What you *can* do is to remember part of the name. This is pretty easy, because the language and library designers have a limited memory (just like you and me) and they tend to name related objects with similar names. So, most of the mapping functions (see Chapter 12) will have `MAP` in their names, the GUI library will probably have `WINDOW` in the names of all the functions, macros, and variables having something to do with windows, and so on.

Once you have a good guess at a part of a name, you can find *all* of the matching names by using a very handy tool named `APROPOS`.

```
? (apropos "MAP" :cl)
MAP, Def: FUNCTION
MAP-INTO, Def: FUNCTION
MAPC, Def: FUNCTION
MAPCAN, Def: FUNCTION
MAPCAR, Def: FUNCTION
MAPCON, Def: FUNCTION
MAPHASH, Def: FUNCTION
MAPL, Def: FUNCTION
MAPLIST, Def: FUNCTION
```

`APROPOS` expects a string or a symbol -- this provides the fragment of the name that you'd like to find. An optional second argument designates a package; use it if you'd like to limit your search to the symbols in a particular package. The package designator can be a string or symbol matching the name or nickname of a package (see Chapter 3, Lesson 10), or it can be the package object itself. If you omit the package designator, then `APROPOS` will search for symbols in *all* packages.

Your Lisp implementation may produce output that looks somewhat different from that shown here. Generally, you'll see the symbol names listed with a very brief description of the global object named by the symbol.

Here's an example of the output from `APROPOS` on my Lisp system when I search without a package designator:

```
? (apropos "SEQUENCE")
       TOOLS::BROWSER-SEQUENCE-TABLE
         CCL::CHECK-SEQUENCE-BOUNDS, Def: FUNCTION
     ITERATE::CHECK-SEQUENCE-KEYWORDS, Def: FUNCTION
       TOOLS::CLASS-BROWSER-SEQUENCE-TABLE
     ITERATE::CLAUSE-FOR-IN-SEQUENCE-0, Def: FUNCTION
     ITERATE::CLAUSE-FOR-INDEX-OF-SEQUENCE-0, Def: FUNCTION
         CCL::CONSED-SEQUENCE
; many lines omitted
              STREAM-READ-SEQUENCE, Def: STANDARD-GENERIC-FUNCTION
              STREAM-WRITE-SEQUENCE, Def: STANDARD-GENERIC-FUNCTION
   DEFSYSTEM::SYSTEMS-SEQUENCE-DIALOG-ITEM
              TAB-SEQUENCE-ITEM
              TABLE-SEQUENCE, Def: STANDARD-GENERIC-FUNCTION
             :TABLE-SEQUENCE,  Value: :TABLE-SEQUENCE
  SETF::|CCL::TABLE-SEQUENCE|, Def: STANDARD-GENERIC-FUNCTION
              WRITE-SEQUENCE, Def: FUNCTION
```

Notice that most of the symbols are prefixed with a package name, since they are not imported into the current (COMMON-LISP-USER) package. Again, your Lisp implementation may produce somewhat different results, but it should show you a list of symbols with package prefixes, plus whatever other information the designers of your implementation thought would be helpful yet concise.

In these examples, I've used uppercase strings as arguments to `APROPOS`. Some implementations perform a case-sensitive match, while others ignore case. Symbol names are case sensitive in Lisp, but the Lisp reader translates symbols to uppercase by default. So if you specify an uppercase name to `APROPOS` you'll find most -- perhaps all -- of the matching names; you'll miss only those that were intentionally interned using lower case letters. And if your `APROPOS` ignores case, you'll get all of the matches, regardless of case.

You could also supply a symbol to `APROPOS`. The only disadvantage of this is that these partial symbols pollute the namespace of the current package. The storage requirement is trivial, but you'll have to be aware that `APROPOS` may find these partial symbols in future matches, and you'll have to ignore these to find the symbols you'd really like to see.

DESCRIBE: Tell me more about yourself

Once you know the name of a symbol, you can get additional information by using the `DESCRIBE` function. As with `APROPOS`, the output of `DESCRIBE` varies among Lisp implementations. Here's an example generated using my Lisp system:

```
; Describe a symbol
? (describe 'length)
Symbol: LENGTH
Function
EXTERNAL in package: #<Package "COMMON-LISP">
Print name: "LENGTH"
Value: #<Unbound>
Function: #<Compiled-function LENGTH #x34C39B6>
Arglist: (SEQUENCE)
Plist: (:ANSI-CL-URL "fun_length.html")
; Describe a string
? (describe "LENGTH")
"LENGTH"
Type: (SIMPLE-BASE-STRING 6)
Class: #<BUILT-IN-CLASS SIMPLE-BASE-STRING>
Length: 6
0: #\L
1: #\E
2: #\N
3: #\G
4: #\T
```

```
5: #\H
; Describe a function
? (describe #'length)
#<Compiled-function LENGTH #x34C39B6>
Name: LENGTH
Arglist (declaration): (SEQUENCE)
```

This example used three different argument types: a symbol, a string, and a function. These are all correct, but you get what you ask for. These all have their uses, but you will generally want to supply a symbol argument to `DESCRIBE`, because it tends to produce the most information.

INSPECT: Open wide and say "Ah..."

`INSPECT` is like `DESCRIBE`, but instead of printing the information it presents the information in some kind of interactive display; typically either a command loop in the current listener or a new window with its own user interface. You should experiment with `INSPECT` on your own Lisp system to learn how it behaves.

`INSPECT` is very handy for exploring complex nested data structures, since you can "drill down" to just the information that interests you at the moment. Most `INSPECT`s offer specialized viewers for certain types of data, such as functions and CLOS objects. Many implementations of `INSPECT` also allow you to edit the data being inspected.

DOCUMENTATION: I know I wrote that down somewhere

Sometimes, you need to know more about a variable than you can discover with `INSPECT`. And for functions, you really need the programmer's description (unless you're willing to read assembly language code, see Chapter 16 if you have these urges). The `DOCUMENTATION` function gives you access to the programmer's innermost thoughts (or at least what she was willing to write in a documentation string).

The `DOCUMENTATION` function expects two arguments. The first is an object for which you wish to retrieve documentation, or a symbol naming that object. The second is a symbol designating the kind of documentation (there are several) you wish to retrieve.

> The interface described above is the one required by the ANSI Common Lisp specification. Some implementations still support an interface which predates the ANSI specification -- these expect a symbol for both arguments. We'll use that convention in our examples, since it works in both ANSI and pre-ANSI implementations.

Several Lisp constructs let you provide an optional documentation string. The following table shows the second `DOCUMENTATION` argument (first line of each pair) together with the Lisp constructs for which documentation strings are retrieved.

`'variable`	`defvar`, `defparameter`, `defconstant`
`'function`	`defun`, `defmacro`, special forms
`'structure`	`defstruct`
`'type`	`deftype`
`'setf`	

```
      defsetf
```

The list above works in both ANSI and pre-ANSI Lisp implementations. The following list shows the documentation types which have been added for ANSI implementations.

```
'compiler-macro
      define-compiler-macro
'method-combination
      define-method-combination
t
```

Documentation returned depends upon type of first argument.

Here are some examples:

```
? (documentation 'length 'function)
"returns the number of elements in sequence."
? (defun a-function-with-docstring ()
    "This function always returns T."
    t)
A-FUNCTION-WITH-DOCSTRING
? (documentation 'a-function-with-docstring 'function)
"This function always returns T."
```

If you specify a symbol for which there is no documentation, or a documentation type which does not select documentation defined for the symbol, then `DOCUMENTATION` will return `NIL`.

A Lisp implementation is permitted to discard documentation strings. If documentation strings from your own program are not accessible via the `DOCUMENTATION` function, consult your implementation's manual to find out whether there's a way to retain documentation strings (there usually is). If documentation strings are missing from Common Lisp functions or from vendor supplied libraries, consult your vendor.

Contents | Cover
Chapter 9 | Chapter 10 | Chapter 11

Copyright © 1995-2001, David B. Lamkins
All Rights Reserved Worldwide

This book may not be reproduced without the written consent of its author. Online distribution is restricted to the author's site.

Chapter 11 - Destructive Modification

Assignment is very different from binding, and in many cases results in programs that are harder to understand. Despite this, there are (sometimes) reasons to prefer assignment. In this chapter, we'll explore assignment and its relationship to destructive modification of data. We'll also explore several Lisp functions that implement destructive modification.

Simple assignment is destructive modification

Any time your program invokes `SETQ` or `SETF`, it is assigning a new value to an existing storage location, destroying the value that was previously in that location. As we'll see in this chapter, there are both risks and benefits to the use of assignment; you need to understand the tradeoffs in order to write Lisp code that is both correct and efficient.

The risk of assignment

Any time you define a function that uses variables, the variables are either `bound` or `free`. A bound variable occurs within a binding form that occurs within the function definition. A binding form is just a form that creates a new association between the name of a variable and a place to store its value; the most common binding forms are `LET` and the argument list of a `DEFUN` or `LAMBDA`.

> There's a slight terminology clash in the use of the word *bound*. The clash is always resolved by the context of the word's use, but you need to be aware of the two meanings. In this chapter we're talking exclusively about a variable *name* being bound to a place to store its value; when we say that Lisp creates a binding for a variable, we mean that it creates a *new* place to store a value under a given name.
>
> The other sense of *bound* -- not otherwise discussed in this chapter -- is the binding of a *value* to a storage location; Lisp supports the notion of an unbound -- or nonexistent -- value.

A variable is *free* within a function if the function provides no binding form for the variable's name. In the following example, the variable `E` is free in both functions `CLOSURE-1` and `CLOSURE-2`.

```
? (let ((e 1))
    (defun closure-1 () e))
CLOSURE-1
? (closure-1)
1
? e
Error: unbound variable
```

So, what happens when a function has to reference a free variable? Lisp creates a *closure* that captures the bindings of free variables for the function. Variables that are free within a function really do have bindings, but the bindings are outside of the function definition. When Lisp executes the function, it finds free variables in the closure. (We'll examine closures in greater detail in Chapter 15.)

Closures are important because they let a function `capture` and retain lexical bindings. Take another look at the example above. When we evaluated `(CLOSURE-1)`, the variable `E` was no longer visible at the top level prompt. But because the function had a closure for that variable, it still has access to its binding.

Let's extend the previous example just a little.

```
? (let ((e 1))
    (defun closure-1 () e)
    (setq e 7)
    (defun closure-2 () e))
CLOSURE-2
? (closure-1)
7
? (closure-2)
7
```

Do you understand why `(CLOSURE-1)` returned 7 rather than 1? We created a binding for the variable `E` and gave it an initial value of 1. Even though `CLOSURE-1` was defined when `E`'s value was 1, this doesn't matter: the closure captures the binding -- the association between the name and the storage location. When we assigned 7 as the value of `E` (just before defining `CLOSURE-2`), we changed only the one storage location for that binding. Since both functions' free variable `E` is closed over the same binding, they must retrieve the same value.

This behavior can be used to good effect.

```
? (let ((counter 0))
    (defun counter-next ()
      (incf counter))
    (defun counter-reset ()
      (setq counter 0)))
COUNTER-RESET
? (counter-next)
1
? (counter-next)
2
? (counter-next)
3
? (counter-next)
4
? (counter-reset)
0
? (counter-next)
1
```

However, some Lisp iteration forms bind their iteration variables just once, then assign new values on subsequent iterations. `DO` and `DO*` assign to their iteration variables. `DOLIST` and `DOTIMES` are allowed to assign to their iteration variables (and probably will in any implementation, because it is more efficient). You need to keep this in mind if you write code that creates a closure for an iteration variable. This example illustrates the point (see Chapter 12 if you want to read about `MAPCAR`):

```
; Closure captures assigned variable -- probably wrong
? (let ((fns ()))
    (dotimes (i 3)
      (push #'(lambda () i) fns))
    (mapcar #'funcall fns))
(3 3 3)
; New bindind created for each captured variable
? (let ((fns ()))
    (dotimes (i 3)
      (let ((i i))
        (push #'(lambda () i) fns)))
    (mapcar #'funcall fns))
(2 1 0)
```

We've seen that assignment can cause unexpected behavior in the presence of closures. Assignment can also cause problems when shared data is involved.

```
? (defun nil-nth (n l)
    "Set nth element of list to nil and return modified list."
    (setf (nth n l) nil)
    l)
NIL-NTH
? (defparameter *my-list* (list 1 2 3 4))
*MY-LIST*
? (nil-nth 1 *my-list*)
(1 NIL 3 4)
? *MY-LIST*
(1 NIL 3 4)
```

> *WARNING:* If you're accustomed to programming in a language that allows by-reference modification of function parameters, the previous code snippet may seem very tantalizing to you. My advice is to put aside all thoughts of using this to emulate by-reference parameters, and use multiple values (Chapter 3, Lesson 9) to safely and efficiently return multiple results from a function.

The above example is not wrong, but it is dangerous. Except in very special situations, we'd like our functions to accept arguments and return values. The problem with `NIL-NTH` is that it assigns a new value within the list passed as a parameter. In our example, this list is global, and may be shared by other parts of the program. If all we really wanted to do was to get a copy of the argument list with the Nth element set to `NIL`, then we shouldn't have altered the passed argument. Here's a better way to implement `NIL-NTH`:

```
? (defun nil-nth (n l)
    "Return list with nth element set to nil."
    (if (zerop n)
      (cons nil (rest l))
      (cons (car l) (nil-nth (1- n) (rest l)))))
NIL-NTH
? (defparameter *my-list* (list 1 2 3 4))
*MY-LIST*
? (nil-nth 1 *my-list*)
(1 NIL 3 4)
? *MY-LIST*
(1 2 3 4)
```

Changing vs. copying: an issue of efficiency

If assignment is so fraught with peril, why not just omit it from the language? There are two reasons: expressiveness and efficiency. Assignment is the clearest way to alter shared data. And assignment is more efficient than binding. Binding creates a new storage location, which allocates storage, which consumes additional memory (if the binding never goes out of scope) or taxes the garbage collector (if the binding eventually does go out of scope).

Modifying lists with destructive functions

Some operations on lists (and sequences -- see Chapter 12) have both destructive and nondestructive counterparts.

Nondestructive	Destructive
SUBLIS	NSUBLIS
SUBST	NSUBST
SUBST-IF	NSUBST-IF
SUBST-IF-NOT	NSUBST-IF-NOT
APPEND	NCONC
REVAPPEND	NRECONC
BUTLAST	NBUTLAST
INTERSECTION	NINTERSECTION
SET-DIFFERENCE	NSET-DIFFERENCE
SET-EXCLUSIVE-OR	NSET-EXCLUSIVE-OR
UNION	NUNION
REVERSE	NREVERSE
REMOVE	DELETE
REMOVE-IF	DELETE-IF
REMOVE-IF-NOT	DELETE-IF-NOT
SUBSTITUTE	NSUBSTITUTE
SUBSTITUTE-IF	NSUBSTITUTE-IF
SUBSTITUTE-IF-NOT	NSUBSTITUTE-IF-NOT
REMOVE-DUPLICATES	DELETE-DUPLICATES

All of these pairings have the same relationship: the destructive version may be faster, but may also alter shared structure. Consider, for example, `APPEND` and `NCONC`. Both append the lists supplied as their arguments.

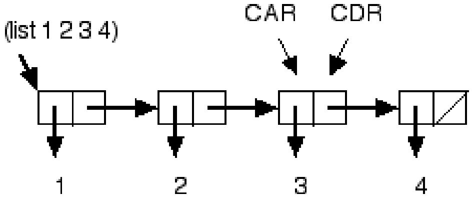
```
? (append (list 1 2 3) (list 4 5 6))
(1 2 3 4 5 6)
? (nconc (list 1 2 3) (list 4 5 6))
(1 2 3 4 5 6)
```

But `NCONC` may destructively modify all but the final list; it may change the tail of each list to point to the head of the next list.

```
? (defparameter list1 (list 1 2 3))
LIST1
? (defparameter list2 (list 4 5 6))
LIST2
? (append list1 list2)
(1 2 3 4 5 6)
? list1
(1 2 3)
? list2
(4 5 6)
? (nconc list1 list2)
(1 2 3 4 5 6)
? list1
(1 2 3 4 5 6) ; Oops - compare to previous result!
? list2
(4 5 6)
```

RPLACA, RPLACD, SETF ...; circularity

A list is constructed of `CONS` cells. Each CONS has two parts, a `CAR` and a `CDR` (review Chapter 3, Lesson 4). The CAR holds the data for one element of the list, and the CDR holds the CONS that makes up the head of the rest of the list.

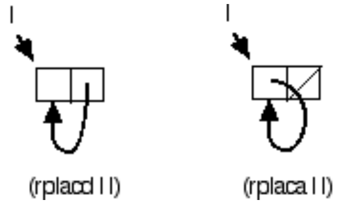


By using `RPLACA` and `RPLACD` to change the two fields of a CONS, we can (destructively) alter the normal structure of a list. For example, we could splice out the second element of a list like this:

```
? (defparameter *my-list* (list 1 2 3 4))
*MY-LIST*
? (rplacd *my-list* (cdr (cdr *my-list*)))
(1 3 4)
? *my-list*
(1 3 4)
```

We can also use these "list surgery operators" to create circular lists.

```
? (let ((l (list 1)))
    (rplacd l l)
    l)
(1 1 1 1 1 1 1 1 ... ; Continues until interrupt or stack overflow
? (let ((l (list 2)))
    (rplaca l l)
    l)
(((((((((((((((( ... ; Continues until interrupt or stack overflow
```



We can get the same effect using `(SETF CAR)` in place of `RPLACA` and `(SETF CDR)` in place of `RPLACD`.

```
(rplaca cons object) is (setf (car cons) object)
(rplacd cons object) is (setf (cdr cons) object)
```

The nice thing about the `SETF` notation is that it readily generalizes to other list accessors, such as `NTH`, `LAST`, `NTHCDR`, and `FOURTH`.

Places vs. values: destructive functions don't always have the desired side-effect

A nondestructive function such as `REVERSE` always returns a freshly constructed result, so there's never any question but that you need to pay attention to the result. But a destructive function such as `NREVERSE` *sometimes* modifies its argument in such a way that the changed argument is identical to the function result. This leads some programmers to assume that destructive functions *always* modify the argument to match the result. Unfortunately, this is not true; leading to the second important point about the use of destructive functions: you should use the result of a destructive function the same way that you would use the result of its nondestructive counterpart.

> This also applies to `SORT` and `STABLE-SORT`, which are destructive and do not have a nondestructive counterpart.

Contrast e.g. PUSH and DELETE

Here's an example showing why you should not depend upon `DELETE`'s side-effects.

```
? (defparameter *my-list (list 1 2 3 4))
*MY-LIST*
? (delete 3 *my-list*)
(1 2 4)
? *my-list*
(1 2 4)
? (delete 1 *my-list*)
(2 4)
? *my-list*
(1 2 4) ; Not the same as function result
```

But some macros, for example `PUSH` and `POP`, take a *place* as an argument and arrange to update the place with the correct value.

```
? (defparameter *stack* ())
*STACK*
? (push 3 *stack*)
(3)
? (push 2 *stack*)
(2 3)
? (push 1 *stack*)
(1 2 3)
? *stack*
(1 2 3)
? (pop *stack*)
1
? *stack*
(2 3)
```

Shared and constant data: Dangers of destructive changes

When you use destructive functions you should be sure to only modify data that your program has constructed at runtime. Here's an example of what can happen if you destructively modify a constant list.

```
? (defun stomp-a-constant ()
    (let ((l '(1 2 3))) ; compile-time constant data
      (print l)
      (setf (second l) nil) ; destructive modification
      l))
STOMP-A-CONSTANT
```

```
? (stomp-a-constant)
(1 2 3)
(1 NIL 3)
? (stomp-a-constant)
(1 NIL 3)
(1 NIL 3)
```

This function is effectively modifying itself, as it changes the constant data which is bound to the variable `L`. The effects of this change show up in the first line of output on the second run (and all subsequent runs).

If you replace `'(1 2 3)` (which may be compiled into constant data) with `(list 1 2 3)` (which always creates a fresh list at run time) then the function's behavior will be identical on the first and all subsequent runs.

Contents | Cover
Chapter 10 | Chapter 11 | Chapter 12

Copyright © 1995-2001, David B. Lamkins
All Rights Reserved Worldwide

This book may not be reproduced without the written consent of its author. Online distribution is restricted to the author's site.

Chapter 12 - Mapping Instead of Iteration

In this chapter we'll survey of a group of functions collectively known as mapping functions. You can think of a mapping function as a kind of special purpose iterator. Every mapping function expects you to supply a function. A typical mapping function applies your function to every element of the supplied list(s). One variation on this theme applies your function to successive sublists.

A sequence is a generalization of the list data type. Vectors (one-dimensional arrays) and lists are specializations of the sequence data type. Some mapping functions work only with lists as inputs, while others accept sequences.

MAPCAR, MAPC, and MAPCAN process successive list elements

The first group of mapping functions processes successive elements of lists. The mapping functions in this group differ in how they construct a return value.

`MAPCAR` processes successive elements of one or more supplied lists. You must supply a function that accepts as many arguments as the number of lists you supply to `MAPCAR`, which applies your function to successive elements and combines the function's results into a freshly constructed list. The mapping stops upon reaching the end of the shortest list; `MAPCAR`'s result has as many elements as the shortest input list.

`MAPC` does not combine the results of applying your function to successive elements of the input list(s). Instead, it processes the inputs just for effect, and returns the first input list as the result of `MAPC`.

`MAPCAN` combines results using the destructive function `NCONC`. Since `NCONC` -- like its nondestructive counterpart `APPEND` -- expects its arguments to be lists, the function you supply to `MAPCAN` must always return a list.

```
? (mapcar #'atom (list 1 '(2) "foo" nil))
(T NIL T T)
? (mapcar #'+ (list 1 2 3) (list 4 5 6))
(5 7 9)
? (mapc #'(lambda (x y) (print (* x y))) (list 1 0 2) (list 3 4 5))

3
0
10
(1 0 2)
? (mapcan #'list (list 1 2 3) (list 4 5 6))
(1 4 2 5 3 6)
? (mapcan #'(lambda (a b) (list (cons a b))) (list 1 2 3) (list 4 5 6))
((1 . 4) (2 . 5) (3 . 6))
```

MAPLIST, MAPL, and MAPCON process successive sublists

`MAPLIST` processes successive sublists of one or more supplied lists. You must supply a function that accepts as many arguments as the number of lists you supply to `MAPLIST`, which applies your function to successive sublists and combines the function's results into a freshly constructed list. The mapping stops upon reaching the end of the shortest list; `MAPLIST`'s result has as many elements as the shortest input list.

`MAPL` does not combine the results of applying your function to successive sublists of the input list(s). Instead, it processes the inputs just for effect, and returns the first input list as the result of `MAPL`.

`MAPCON` combines results using the destructive function `NCONC`. Since `NCONC` -- like its nondestructive counterpart `APPEND` -- expects its arguments to be lists, the function you supply to `MAPCON` must always return a list.

```
? (maplist #'list (list 1 2 3) (list 4 5 6))
(((1 2 3) (4 5 6)) ((2 3) (5 6)) ((3) (6)))
? (mapl #'(lambda (x y) (print (append x y))) (list 1 0 2) (list 3 4 5))

(1 0 2 3 4 5)
(0 2 4 5)
(2 5)
(1 0 2)
? (mapcon #'list (list 1 2 3) (list 4 5 6))
((1 2 3) (4 5 6) (2 3) (5 6) (3) (6))
```

MAP and MAP-INTO work on sequences, not just lists

A sequence is either a list or a vector (a one-dimensional array). The previous group of mapping functions (`MAPCAR` et al) processes successive CARs or CDRs of their input lists. `MAP` and `MAP-INTO` process successive elements of their input sequences.

`MAP` requires that you specify the type of its result using one of the following designators:

```
Designator   Result
----------   ------
NIL          NIL
'LIST        a list
'VECTOR      a vector
```

Note that you can also specify subtypes of `LIST` or `VECTOR` -- your Lisp implementation may be able to optimize the storage of the result based on the type you specify.

```
? (map nil #'+ (list 1 2 3) (list 4 5 6))
NIL
? (map 'list #'+ (list 1 2 3) (list 4 5 6))
(5 7 9)
? (map 'vector #'+ (list 1 2 3) (list 4 5 6))
#(5 7 9)
? (map '(vector number 3) #'+ (list 1 2 3) (list 4 5 6))
#(5 7 9)
```

`MAP-INTO` is a destructive version of `MAP`. The first argument is a sequence that receives the results of the mapping. Mapping stops upon reaching the end of the result sequence or any of the input sequences. (Therefore, if you specify `NIL` as the result sequence, no mapping is performed since `NIL` is a list of length zero.) The input sequences are not modified. The modified result sequence is returned as the value of `MAP-INTO`.

```
? (let ((a (make-sequence 'list 3)))
    (print a)
    (map-into a #'+ (list 1 2 3) (list 4 5 6))
    a)
```

```
(NIL NIL NIL)
(5 7 9)
? (let ((a (make-sequence 'vector 3)))
    (print a)
    (map-into a #'+ (list 1 2 3) (list 4 5 6))
    a)

#(0 0 0)
#(5 7 9)
```

Your Lisp implementation may print different initial values for the unmodified sequences in the above examples. If you need to specify a certain initial value for `MAKE-SEQUENCE`, use the `:INITIAL-ELEMENT` keyword argument:

```
? (let ((a (make-sequence 'list 3 :initial-element 0)))
    (print a)
    (map-into a #'+ (list 1 2 3) (list 4 5 6))
    a)

(0 0 0)
(5 7 9)
```

Mapping functions are good for filtering

A filter passes some of its inputs through to its output, and drops others. We can use mapping functions to implement filters by taking note of the behavior of `APPEND`:

```
? (append '(1) nil '(3) '(4))
(1 3 4)
```

The `NIL` argument (which is equivalent to the empty list) simply "disappears" from the result. This is the key observation that we need to construct a filter. We'll use `MAPCAN` to map over our input list(s) and supply a mapping function that:

- makes a list of each result we wish to retain in the output, or
- returns `NIL` in place of each input we wish to exclude from the output.

```
? (defun filter-even-numbers (numbers)
    (mapcan #'(lambda (n) (when (evenp n) (list n))) numbers))
FILTER-EVEN-NUMBERS
? (filter-even-numbers (list 1 2 3 4 5 6 7 8))
(2 4 6 8)
```

> `WHEN` returns `NIL` if the condition is `NIL`. We could have written `(if (evenp n) (list n) nil)` instead.

Here's a slightly more complex filter that returns a list of evenly divisible pairs of numerators and denominators:

```
? (defun filter-evenly-divisible (numerators denominators)
    (mapcan #'(lambda (n d)
                (if (zerop (mod n d))
                  (list (list n d))
                  nil))
            numerators denominators))
```

```
? (filter-evenly-divisible (list 7 8 9 10 11 12)
                           (list 1 4 5  5  2  3))
((7 1) (8 4) (10 5) (12 3))
```

The functions `REMOVE-IF` and `REMOVE-IF-NOT` (and their destructive counterparts, `DELETE-IF` and `DELETE-IF-NOT`) filter a single sequence, but can't be used for multiple sequences (as in the example above). Use `REMOVE-IF` and the like if it will make your code clearer. See Chapter 13 for further details.

It's better to avoid mapping if you care about efficiency

Most Lisp systems will generate more efficient code to call a function that is known during compilation than a function that can change at run time. Mapping functions accept a functional argument, and most compilers will generate code that supports run time function binding -- even if you specify a "constant" function, such as `#'+`. Also, the run time call may incur extra overhead to generate a list of arguments for the function's application.

Therefore, if you are concerned about efficiency you should write map-like functions using iteration instead of mapping functions. But do this only when you are sure that efficiency is an issue for the portion of the program you intend to rewrite. See Chapter 28 for a discussion of profiling, which can help you find your program's performance bottlenecks.

Predicate mapping functions test sequences

Sometimes you may need to apply a test to some input sequences and return a truth value based upon what the test returned for all of the inputs. For example, you might want to know whether any number in a sequence is outside of a specified range, or whether every word is at least five letters long. You could construct these tests from the mapping functions described above, but that would be more verbose (and less efficient) than using the predicate mapping functions provided by Lisp.

SOME, EVERY, NOTANY, NOTEVERY

The built in predicate mapping functions expect you to supply a test function (a.k.a. predicate) and one or more input sequences. The predicate is applied to successive elements of the input sequences until the the result of the mapping function can be determined.

```
Function   Condition
--------   ---------
SOME       user-supplied predicate succeeds on at least one input
EVERY      user-supplied predicate succeeds on every input
NOTANY     complement of SOME
NOTEVERY   complement of EVERY
```

For example, `SOME` examines inputs so long as the predicate is false; the tests stop -- and `SOME` returns a true value -- as soon as the predicate is true for some input(s). If the predicate is false for every input, `SOME` returns a false value.

Similarly, `EVERY` examines inputs so long as the predicate is true; the tests stop -- and `EVERY` returns a false value -- as soon as the predicate is false for some input(s). If the predicate is true for every input, `EVERY` returns a true value.

```
? (some #'(lambda (n) (or (< n 0) (> n 100))) (list 0 1 99 100))
```

```
NIL
? (some #'(lambda (n) (or (< n 0) (> n 100))) (list -1 0 1 99 100))
T
? (every #'(lambda (w) (>= (length w) 5)) (list "bears" "bulls" "racoon"))
T
? (every #'(lambda (w) (>= (length w) 5)) (list "bears" "cat" "racoon"))
NIL
```

And of course, the predicate mapping functions handle multiple sequences as you'd expect.

```
? (some #'> (list 0 1 2 3 4 5) (list 0 0 3 2 6))
T
```

REDUCE combines sequence elements

While we're on the subject of mapping, wouldn't it be nice to be able to combine all of the elements of a sequence using some function? `REDUCE` does just that, accepting a function (of two or zero arguments) and a sequence. If the sequence is longer than one element, `REDUCE` combines the results of applying the function to successive elements of the sequence. For example:

```
? (reduce #'* (list 1 2 3 4 5)) (* (* (* (* 1 2) 3) 4) 5)
120
? (reduce #'- (list 10 2 3 1)) ; (- (- (- 10 2) 3) 1)
4
```

If the sequence is of length one, `REDUCE` returns the sequence and the function is not applied. If the sequence is of length zero, `REDUCE` applies the function with no arguments and returns the value returned by the function. (In the case of arithmetic functions, this is the identity value for the operation.)

Various keyword arguments let you specify a subsequence for `REDUCE`, or that `REDUCE` should combine elements in a right-associative manner (i.e. from the end of the sequence, rather than from the beginning).

Contents | Cover
Chapter 11 | Chapter 12 | Chapter 13

Copyright © 1995-2001, David B. Lamkins
All Rights Reserved Worldwide

This book may not be reproduced without the written consent of its author. Online distribution is restricted to the author's site.

Chapter 13 - Still More Things You Can Do with Sequences

In this chapter we'll meet the most useful sequence functions, and see how to use them. We'll also reprise earlier admonitions about proper use of destructive functions.

CONCATENATE: new sequences from old

`CONCATENATE` always creates a new sequence from (of course) the concatenation of zero or more argument sequences. You must specify the type of the result, and the argument types must be proper subtypes of the sequence type.

```
? (concatenate 'list) ; no argument sequences
NIL
? (concatenate 'vector) ; no argument sequences
#()
? (concatenate 'list '(1 2 3) '(4 5))
(1 2 3 4 5)
? (concatenate 'vector #(1 2 3) #(4 5))
#(1 2 3 4 5)
? (concatenate 'list #(1 2 4) '(4 5))
(1 2 3 4 5)
? (concatenate 'vector '(1 2 3) #(4 5))
#(1 2 3 4 5)
? (concatenate 'list "hello") ; string is a subtype of sequence
(#\h #\e #\l #\l #\o)
```

ELT and SUBSEQ get what you want from any sequence (also, COPY-SEQ)

If you need to pick out one element (or a range of elements) from a sequence, you can use `ELT` (to pick out one element) or `SUBSEQ` (to pick out a range of elements). But don't use these unless you're really sure you can't narrow down the sequence type to a vector or list; there are more specific (hence more efficient) accessors for the less general types.

`SUBSEQ` makes a copy of a specified portion of a sequence. `COPY-SEQ` is closely related to `SUBSEQ`, except that it copies *all* of the elements of a sequence.

```
? (elt '(1 2 3 4 5) 1) ; zero-based indexing
2
? (subseq '(1 2 3 4 5) 2) ; 3rd element through end
(3 4 5)
? (let ((l '(1 2 3 4 5)))
    (subseq l 2 (length l))) ; same effect as previous
? (subseq '(1 2 3 4 5) 0 3) ; element at ending index is not copied
(1 2 3)
? (subseq #(#\a #\b #\c #\d #\e) 2 4)
#(#\c #\d)
? (copy-seq '(a b c))
(A B C)
```

REVERSE turns a sequence end-for-end (also, NREVERSE)

`REVERSE` makes a copy of a sequence, with the order of elements reversed. `NREVERSE` is the destructive counterpart of `REVERSE`; it is more efficient, but it modifies its input argument.

`REVERSE` is commonly used in code similar to the following.

```
(defun collect-even-numbers (number-list)
  (let ((result ()))
    (dolist (number number-list)
      (when (evenp number)
        (push number result)))
    (nreverse result)))
```

The `DOLIST` and `PUSH` collect even numbers on the result list, but they are in the reverse order of their original positions on the input list. The final `NREVERSE` puts them back into their original order. This is a safe use of the destructive function `NREVERSE` because the `RESULT` variable can not be shared; it is forgotten as soon as control leaves the `LET` form.

LENGTH: size counts after all

There's not much to say about `LENGTH`. Just remember that for lists, `LENGTH` counts only the elements of the top-level list, and not those of any nested lists.

```
? (length '((1 2 3) (4 5) (6) 7 () 8 9))
7
```

COUNT: when it's what's inside that matters

If you find your program filters a sequence only to get the length of the result, use `COUNT` (and related functions `COUNT-IF` and `COUNT-IF-NOT`) instead.

```
? (count 3 '(1 3 3 4 2 5 9 8 3 1 9)) ; count occurrences
3
? (count-if #'oddp '(1 3 3 4 2 5 9 8 3 1 9)) ; count matches to predicate
8
? (count-if-not #'evenp '(1 3 3 4 2 5 9 8 3 1 9)) ; count mismatches using predicate
8
```

These functions accept keyword arguments:

Keyword	Value	Default
:START	starting index (inclusive)	0
:END	ending index (exclusive)	NIL
:FROM-END	non-NIL to work backwards from end element	NIL
:KEY	function to select match data from element	NIL

A `NIL` value for the `:END` keyword designates a position just past the end of the sequence; since this is an exclusive limit, the last element will be processed. (If you specified the index of the last element, the last element would *not* be processed.)

The `:FROM-END` keyword is useful in the case that the test function has side-effects, and the order of the side-effects is important.

When the `:KEY` argument is not `NIL`, it should be a function of one argument that extracts data from the sequence element. For example:

```
? (count 3 '((1 2 3) (2 3 1) (3 1 2) (2 1 3) (1 3 2) (3 2 1)) :key #'second)
2
```

`COUNT` accepts the additional keyword arguments `:TEST` and `:TEST-NOT`. These give you a compact way to write a test that involves a second value. Compare the following equivalent forms:

```
; Using COUNT-IF and LAMBDA
(count-if #'(lambda (n) (<3 n)) '(1 2 3 4 5 6 7))

; Using COUNT and :TEST
(count 3 '(1 2 3 4 5 6 7) :test #'
```

The keyword arguments for comparison predicates also let you define the precise meaning of equality. The default predicate is `EQL`, which is true for identical numbers and symbols. See Chapter 17 for more information on comparison predicates.

REMOVE, SUBSTITUTE, and other sequence changers

`REMOVE` removes all occurrences of a specified element from a sequence.

```
? (remove 7 '(1 2 3 a b c t nil 7 0 7 7))
(1 2 3 A B C T NIL 0)
```

Keyword arguments are handled in the same way as for `COUNT`. `REMOVE-IF` and `REMOVE-IF-NOT` are also available; their keyword arguments are handled in the same way as for `COUNT-IF` and `COUNT-IF-NOT`.

A `:COUNT` keyword argument lets you limit the number of matching elements to remove.

`SUBSTITUTE` changes all occurrences of a specified element in a sequence to another value.

```
? (substitute '(q) 7 '(1 2 3 a b c t nil 7 0 7 7))
(1 2 3 A B C T NIL (Q) 0 (Q) (Q))
```

Keyword arguments are handled in the same way as for `COUNT`. `SUBSTITUTE-IF` and `SUBSTITUTE-IF-NOT` are also available; their keyword arguments are handled in the same way as for `COUNT-IF` and `COUNT-IF-NOT`.

A `:COUNT` keyword argument lets you limit the number of matching elements to substitute.

`REMOVE-DUPLICATES` returns a copy of a sequence, modified so that every element is different.

```
? (remove-duplicates '(1 2 3 a b c (1 2 3) f c g c h b i a j b a k a))
(1 2 3 (1 2 3) F G C H I J B K A)
```

The last copy of each identical element is retained in the result, unless you specify the keyword argument `:FROM-END T`, which causes the first copy of each identical element to be retained.

`REMOVE-DUPLICATES` also accepts the same keyword arguments as `COUNT`. The `:TEST` and `:TEST-NOT` keyword arguments let you specify the comparison predicate used to determine whether elements are identical. The default predicate is `EQL`, which is true for identical numbers and symbols. See Chapter 17 for more information on comparison predicates.

DELETE, REMOVE-DUPLICATES, DELETE-DUPLICATES, and NSUBSTITUTE.

Many of the functions in the preceeding section have destructive counterparts. The result of the destructive functions is identical, but the input sequence may be destructively modified.

Nondestructive	Destructive
REMOVE	DELETE
REMOVE-IF	DELETE-IF
REMOVE-IF-NOT	DELETE-IF-NOT
SUBSTITUTE	NSUBSTITUTE
SUBSTITUTE-IF	NSUBSTITUTE-IF
SUBSTITUTE-IF-NOT	NSUBSTITUTE-IF-NOT
REMOVE-DUPLICATES	DELETE-DUPLICATES

> Remember that you *must not* depend upon the modification of the input sequences. The only result guaranteed to be correct is the return value of the function.

FILL and REPLACE

`FILL` destructively modifies a sequence, replacing every element with a new value. It accepts keyword arguments for `:START` and `:END` positions; these have the same meaning as described earlier in this chapter. The modified sequence is returned as the value of `FILL`.

```
? (fill (list 1 1 2 3 5 8) 7)
(7 7 7 7 7 7)
? (fill (list 1 1 2 3 5 8) '(a b))
((A B) (A B) (A B) (A B) (A B) (A B))
? (fill (list 1 1 2 3 5 8) 7 :start 2 :end 4)
(1 1 7 7 5 8)
```

`REPLACE` copies elements from one sequence into another, destructively modifying the target sequence. You can specify the range of elements to use in both sequences; the shorter of the two ranges determines the number of elements that is actually copied.

```
? (let ((a (list 1 2 3 4 5 6 7))
        (b (list 9 8 7 6 5 4 3)))
    (replace a b))

(9 8 7 6 5 4 3)
? (let ((a (list 1 2 3 4 5 6 7))
        (b (list 9 8 7 6 5 4 3)))
    (replace a b :start1 2))

(1 2 9 8 7 6 5)
```

```
? (let ((a (list 1 2 3 4 5 6 7))
        (b (list 9 8 7 6 5 4 3)))
    (replace a b :start1 2 :end1 5))
(1 2 9 8 7 6 7)
? (let ((a (list 1 2 3 4 5 6 7))
        (b (list 9 8 7 6 5 4 3)))
    (replace a b :start1 2 :end1 5 :start2 3))
(1 2 6 5 4 6 7)
? (let ((a (list 1 2 3 4 5 6 7))
        (b (list 9 8 7 6 5 4 3)))
    (replace a b :start1 2 :end1 5 :start2 3 :end2 4))
(1 2 6 4 5 6 7)
```

Locating things in sequences: POSITION, FIND, SEARCH, and MISMATCH

`POSITION` searches a sequence for a matching element, and returns the index of the first match or `NIL` if no matching element is in the sequence.

```
? (position #\a "This is all about you, isn't it?")
8
? (position #\! "This is all about you, isn't it?")
NIL
```

`POSITION` accepts the same keyword arguments as `COUNT` (described earlier in this chapter) and has (the by now familar) variants `POSITION-IF` and `POSITION-IF-NOT`.

`FIND` is similar to `POSITION` except that the matching element -- rather than its index in the sequence -- is returned if there is a match. As with `POSITION`, you'll find the usual keyword arguments (`:FROM-END`, `:START`, `:END`, `:KEY` -- and for the "base" function, `:TEST` and `:TEST-NOT`) and function variants (i.e. `FIND-IF` and `FIND-IF-NOT`).

```
? (find #\a "This is all about you, isn't it?")
#\a
? (find #\! "This is all about you, isn't it?")
NIL
```

`SEARCH` returns the starting position of one sequence within another sequence, or `NIL` if no match is found.

```
? (search "ab" "This is all about you, isn't it?")
12
? (search "not so" "This is all about you, isn't it?")
NIL
```

`SEARCH` accepts `:FROM-END`, `:KEY`, `:TEST` and `:TEST-NOT` keyword arguments with the usual interpretations. You can specify a range in the substring (the first argument) using `:START1` and `:END1` keywords, and in the target string using the `:START2` and `:END2` keywords.

`MISMATCH` is the functional complement to `SEARCH` -- it returns the first position at which the substring *fails* to match a portion of the target string.

```
? (mismatch "banana" "bananananono")
6
? (mismatch "." "...hello")
1
```

```
? (mismatch "............" "...hello")
3
```

SORT and MERGE round out the sequence toolkit

`SORT` destructively sorts a sequence; the order is determined by a predicate which you supply.

```
? (sort (list 9 3 5 4 8 7 1 2 0 6) #'>)
(9 8 7 6 5 4 3 2 1 0)
? (sort (list 9 3 5 4 8 7 1 2 0 6) #'<)
(0 1 2 3 4 5 6 7 8 9)
```

The input sequence is destructively modified -- you must use the function result.

`STABLE-SORT` preserves the original order of identical elements; `SORT` may not.

You can sort structured elements (e.g. lists, structures) by using the `:KEY` keyword argment to specify a key extraction function.

`MERGE` combines *two* input sequences into a single result. Elements are interleaved according to the predicate. Either input sequence may be destructively modified. You must designate the type of the result.

```
? (merge 'vector (list 1 3 5 9 8) (vector 2 6 4 7 0) #'>)
#(2 6 4 7 1 3 5 9 8 0)
? (merge 'list (list 1 3 5 9 8) (vector 2 6 4 7 0) #'<)
(1 2 3 5 6 4 7 0 9 8)
? (merge 'vector (list 1 3 5 8 9) (vector 0 2 4 6 7) #'>)
#(1 3 5 8 9 0 2 4 6 7)
? (merge 'list (list 1 3 5 8 9) (vector 0 2 4 6 7) #'<)
(0 1 2 3 4 5 6 7 8 9)
```

Note that -- in the general case -- `MERGE` does *not* sort the catenation of its arguments. The predicate is used to select from one or the other of the input sequences; input from the selected sequence continues until the sense of the predicate changes. Look at the examples until you understand this.

`MERGE` accepts a `:KEY` keyword argument having the conventional meaning.

Contents | Cover
Chapter 12 | Chapter 13 | Chapter 14

Copyright © 1995-2001, David B. Lamkins
All Rights Reserved Worldwide

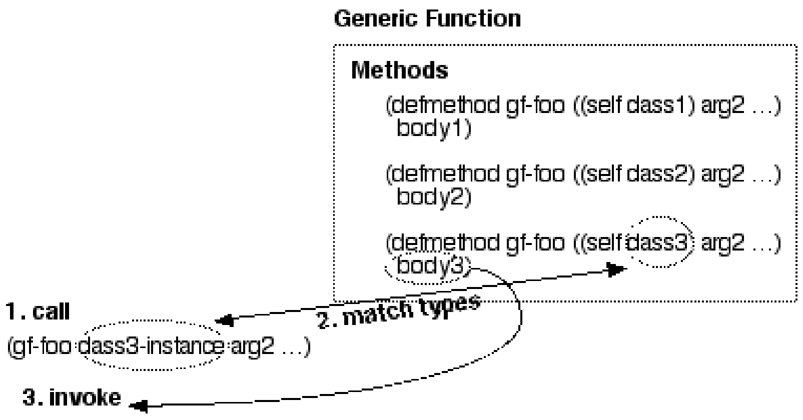
This book may not be reproduced without the written consent of its author. Online distribution is restricted to the author's site.

Chapter 14 - Can Objects Really Behave Themselves?

This chapter continues the treatment of CLOS (the Common Lisp Object System) that we began in Chapter 7, in which we saw how objects store data. In this chapter we'll learn about how objects get their behaviors.

Generic functions give objects their behaviors

A generic function provides behavior based upon the type of an object. The behavior is selected according to the types of the arguments to the generic function. The generic function dispatches control to a particular method that provides the best match to the argument types that you use to invoke the generic function.


Generic Function
Methods
(defmethod gf-foo ((self class1) arg2 ...)
body1)
(defmethod gf-foo ((self class2) arg2 ...)
body2)
(defmethod gf-foo ((self class3) arg2 ...)
body3)
1. call
(gf-foo class3-instance arg2 ...)
2. match types
3. invoke

You define a method using Lisp's `DEFMETHOD` macro. In its simplest form, `DEFMETHOD` expects a name and a *specialized* lambda list. The specialized lambda list is similar to the list of formal parameters you supply for a `LAMBDA` or `DEFUN` form; the difference is that you can specify the type of each parameter. The method will only be invoked when the generic function call (which looks *exactly* like a function call) specifies parameters that are of matching types. To specialize a parameter in a `DEFMETHOD` form, simply name the parameter and its type in a list. For example:

```
(defmethod method1 ((param1 number) (param2 string)) ...)
(defmethod method2 ((param2 float) (param2 sequence)) ...)
```

You can also leave the type of a parameter unspecified by just giving its name; this kind of parameter will match *any* data type. In this example, the first parameter is not specialized:

```
(defmethod method3 (param1 (param2 vector)) ...)
```

Note that the parameter types do not *have* to be CLOS class types. If you *want* to specialize a method to one particular CLOS class, you can specialize one of the arguments to that class, as we saw in the first figure in this chapter. If you have specialized one parameter to a CLOS class, and leave the other parameters unspecialized, then you've emulated the single-dispatch method common to certain "classic" object-based programming languages. In this limited case you can think of a method as being associated with a class. We'll see in the next section that this association breaks down when we associate a method with multiple classes.

You're probably wondering how generic functions get created, when all you do is to define methods. When you define a method, Lisp creates a generic function if one does not already exist. When Lisp creates this generic function for you, it makes note of the name, the number of required and optional arguments, and the presence and names of keyword parameters. When you create another method of the same name, it must agree with the generic function on the details of the parameters which were recorded in the generic function. This agreement is called *lambda list congruence* -- Lisp will signal an error if you attempt to create a new method with a non-congruent lambda list.

The line between methods and objects blurs for multimethods

A multimethod is a method that is selected based upon the types of two or more of its arguments. When you have a method that is selected for two or more classes, then we can't really say that a class (or an object, which is an instance of that class) "has" a particular method.

Methods on non-objects? So where does the method live?

The association of methods to classes gets even more tenuous when we consider that one or more of a method's arguments may specialize on an object that is not a class. If we expect to find some convenient way to say that a method "belongs to" a particular class, we're simply not going to find it.

Simpler (and less capable) object systems than CLOS *do* associate a method with a particular class. CLOS does not. This is an important point, so let me rephrase it: CLOS methods are *not* a part of any class for which they may provide services.

A method is a part of a generic function. The generic function analyzes the actual parameters and selects a method to invoke based upon a match between actual parameters and specialized lambda lists in the method definitions. And, to reiterate the point made in the previous paragraph, generic functions are *not a part of* of any classes upon which the generic function's methods operate.

Generic functions work by dispatching on argument specializers

When you define a method, the types of its parameters (in the specialized lambda list) declare that the method may be invoked only by parameters of the same, or more specific, types. For example, if a parameter is specialized on the type `NUMBER`, it can match `INTEGER`, `FIXNUM`, `FLOAT`, `BIGNUM`, `RATIONAL`, `COMPLEX`, or any other proper subtype of `NUMBER`.

But what if you define two methods that could match the same types? Consider the following definitions.

```
(defmethod op2 ((x number)  (y number))  ...) ; method 1
(defmethod op2 ((x float)   (y float))   ...) ; method 2
(defmethod op2 ((x integer) (y integer)) ...) ; method 3
(defmethod op2 ((x float)   (y number))  ...) ; method 4
(defmethod op2 ((x number)  (y float))   ...) ; method 5
```

A call of the form `(OP2 11 23)` potentially matches methods 1 and 3, because the arguments are both of type *INTEGER*, which is a subtype of `NUMBER`. CLOS resolves ambiguity by choosing the more specific match, thus method 3 is selected for a call of `(OP2 11 23)`.

The same resolution strategy chooses method 5 for `(OP2 13 2.9)`, method 4 for `(OP2 8.3 4/5)`, and method 1 for `(OP2 5/8 11/3)`. The general rule is that CLOS selects a method based upon the most specific matching types, and an argument is always more specific than the arguments to its right. The second part of this rule means that arguments on the left serve as tiebreakers for those futher to the right. Consider these methods:

```
(defmethod Xop2 ((x number)  (y number))  ...) ; method 1
```

```
(defmethod Xop2 ((x float)   (y number))  ...) ; method 2
(defmethod Xop2 ((x number)  (y float))   ...) ; method 3
```

A call of `(XOP2 5.3 4.1)` will invoke method 2. Both method 2 and method 3 are more specific than method 1. Method 2 has a more specialized type in the first argument position when compared to method 3, so method 2 is the one that is invoked.

In addition to dispatching based upon argment types, CLOS can dispatch based upon specific objects.

```
? (defmethod idiv ((numerator integer) (denominator integer))
    (values (floor numerator denominator)))
#<STANDARD-METHOD IDIV (INTEGER INTEGER)>
? (defmethod idiv ((numerator integer) (denominator (eql 0)))
    nil)
#<STANDARD-METHOD IDIV (INTEGER (EQL 0))>
? (idiv 4 3)
1
? (idiv 6 2)
3
? (idiv 4 0)
NIL
```

Here we've specialized on the integer 0. You can specialize on any object that can be distinguished using the `EQL` predicate. Numbers, symbols and object instances can all be tested in this way. See Chapter 17 for more information on the `EQL` predicate.

Object inheritance matters after all; finding the applicable method

A class is a type, and a subclass is a subtype. So when you define these classes:

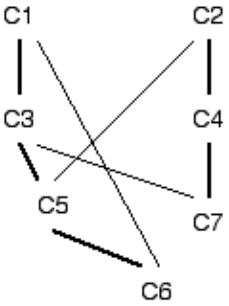
```
(defclass c1 () ...)
(defclass c2 (c1) ...)
```

then `C2` is a subclass of `C1`. If you then use the types `C1` and `C2` as specializers in a method definition, `C2` will be a more specific type (see the previous section) than `C1`.

But what if you define classes that inherit from more than one class? How do you decide which class is more specific than another? Consider the following class definitions:

```
(defclass c1 () ...)
(defclass c2 () ...)
(defclass c3 (c1) ...)
(defclass c4 (c2) ...)
(defclass c5 (c3 c2) ...)
(defclass c6 (c5 c1) ...)
(defclass c7 (c4 c3) ...)
```

These definitions give us an inheritance hierarchy that looks like this; bold lines link a subclass to the first parent class, while lighter lines link to the second parent class:



Now consider the following method definitions, which specialize on this class hierarchy:

```
(defmethod m1 ((x c1)) ...) ; method 1
(defmethod m1 ((x c2)) ...) ; method 2
```

It's clear that calling `M1`'s generic function with an object of type `C1` or `C3` will invoke method 1, and that calling the generic function with an object of type `C2` or `C4` will invoke method 2. But what happens if we call `M1`'s generic function with an object of type `C5`, `C6`, or `C7`? These classes all inherit -- directly or indirectly -- from *both* `C1` and `C2`.

If we invoke the generic function `M1` with an object of type `C5`, `C6`, or `C7`, CLOS must decide whether to invoke method 1 or method 2. (It can't do both.) This decision is based upon some measure of whether `C1` or `C2` is a more specific parent class. The measure is based upon the position of the parent class in the *class precedence list* of the subclass. Here are the class precedence lists (CPLs) for `C5`, `C6`, and `C7`:

```
Class   CPL
-----   ---
C5      (C5 C3 C1 C2)
C6      (C6 C5 C3 C1 C2)
C7      (C7 C4 C2 C3 C1)
```

Classes near the beginning of the CPL are more specific, so `C5` and `C6` are more specific to `C1` and `C7` is more specific to `C2`. Therefore, calling the `M1` generic function with an object of type `C5` or `C6` will invoke method 1. Calling `M1` with an object of type `C7` will invoke method 2.

The next question you should ask is "how *does* CLOS determine the CPL?" There is, of course, an algorithm for computing the CPL -- you can find this described in a Lisp reference manual. Or you can define some classes and ask Lisp to tell you the CPL; most implementations include a function named `CLASS-PRECEDENCE-LIST` that expects a class *object* as its only argument (use `FIND-CLASS` to get the class object from its name) and returns a CPL.

```
? (class-precedence-list (find-class 'c6))
(C6 C5 C3 C1 C2)
```

Design conservatively with multiple inheritance, and you shouldn't have to depend upon knowledge of the algorithm by which CLOS computes the CPL.

Method combinations offer further choices

If you define methods as we've seen throughout this chapter, the generic function that gets created will offer a capability called *standard* method combination. The methods that we've so far used have all been *primary* methods. Under standard method combination, we can also define *before*, *after*, and *around* methods which get combined with the primary method.

To define a *before*, *after*, or *around* method we add a corresponding keyword (a *method qualifier*) to our `DEFMETHOD` form, like this:

```
(defmethod madness :before (...) ...)
(defmethod madness :after  (...) ...)
(defmethod madness :around (...) ...)
```

Let's take a look at standard method combination in action. We'll begin with the `:BEFORE` and `:AFTER` methods.

```
; Define a primary method
? (defmethod combo1 ((x number)) (print 'primary) 1)
#<STANDARD-METHOD COMBO1 (NUMBER)>
; Define before methods
? (defmethod combo1 :before ((x integer)) (print 'before-integer) 2)
#<STANDARD-METHOD COMBO1 :BEFORE (INTEGER)>
? (defmethod combo1 :before ((x rational)) (print 'before-rational) 3)
#<STANDARD-METHOD COMBO1 :BEFORE (RATIONAL)>
; Define after methods
? (defmethod combo1 :after ((x integer)) (print 'after-integer) 4)
#<STANDARD-METHOD COMBO1 :AFTER (INTEGER)>
? (defmethod combo1 :after ((x rational)) (print 'after-rational) 5)
#<STANDARD-METHOD COMBO1 :AFTER (RATIONAL)>
; Try it
? (combo1 17)

BEFORE-INTEGER
BEFORE-RATIONAL
PRIMARY
AFTER-RATIONAL
AFTER-INTEGER
1
? (combo 4/5)

BEFORE-RATIONAL
PRIMARY
AFTER-RATIONAL
1
```

When we call `COMBO1`, CLOS determines which methods are applicable. As we learned earlier, only one primary method is applicable. But, as we saw in the call to `(COMBO1 17)`, we can have multiple applicable `:BEFORE` and `:AFTER` methods. Because integer is a subtype of rational, an integer argument to `COMBO1`, the `:BEFORE` and `:AFTER` methods that specialize on `INTEGER` and `RATIONAL` arguments are applicable.

So CLOS has now determined a set of applicable methods: a primary method and some before and after methods. The standard method combination determines the order in which these methods get invoked. First, *all* of the applicable `:BEFORE` methods are invoked, with the more specific methods invoked *first*. Then the applicable primary method is invoked. Next, all of the applicable `:AFTER` methods are invoked, with the more specific methods invoked *last*. Finally, the value of the primary method is returned as the value of the generic function.

`:BEFORE` and `:AFTER` methods are often used to add extra behaviors to a method. They typically introduce some kind of side effect -- by doing I/O, by changing global state, or by altering slots of one or more of the objects passed as parameters. There are three actions not available to `:BEFORE` and `:AFTER` methods:

1. They can't alter the parameters seen by other applicable methods.
2. They can't alter which of the applicable methods are actually invoked.
3. They can't alter the value returned from the generic function.

But the standard method combination offers a third kind of qualified method, the `:AROUND` method, that can perform all of these actions. An `:AROUND` method is defined using a method qualifier, just as you might expect:

```
(defmethod madness :around (...) ...)
```

When a generic function has one or more `:AROUND` methods defined, the most specific applicable `:AROUND` method is invoked first, even if there are applicable `:BEFORE` methods. At this point, the `:AROUND` method has complete control -- if it simply returns, then *none* of the other applicable methods will be invoked. Normally an `:AROUND` method calls `CALL-NEXT-METHOD` which allows control to proceed through other applicable methods.

`CALL-NEXT-METHOD` calls the next most specific `:AROUND` method; if there are no less specific applicable `:AROUND` methods, then `CALL-NEXT-METHOD` invokes all of the applicable before, primary, and after methods exactly as detailed above. The value returned by the most specific `:AROUND` method is returned by the generic function; typically you'll use the value -- or some modification thereof -- returned by `CALL-NEXT-METHOD`.

If you call `CALL-NEXT-METHOD` without arguments, it uses the arguments of the current method. But you can call `CALL-NEXT-METHOD` with arguments, and change the parameters seen by the called method(s).

```
; Define a primary method
? (defmethod combo2 ((x number)) (print 'primary) 1)
#<STANDARD-METHOD COMBO2 (NUMBER)>
; Define before methods
? (defmethod combo2 :before ((x integer)) (print 'before-integer) 2)
#<STANDARD-METHOD COMBO2 :BEFORE (INTEGER)>
? (defmethod combo2 :before ((x rational)) (print 'before-rational) 3)
#<STANDARD-METHOD COMBO2 :BEFORE (RATIONAL)>
; Define after methods
? (defmethod combo2 :after ((x integer)) (print 'after-integer) 4)
#<STANDARD-METHOD COMBO2 :AFTER (INTEGER)>
? (defmethod combo2 :after ((x rational)) (print 'after-rational) 5)
#<STANDARD-METHOD COMBO2 :AFTER (RATIONAL)>
; Define around methods
? (defmethod combo2 :around ((x float))
    (print 'around-float-before-call-next-method)
    (let ((result (call-next-method (float (truncate x)))))
      (print 'around-float-after-call-next-method)
      result))
#<STANDARD-METHOD COMBO2 :AROUND (FLOAT)>
? (defmethod combo2 :around ((x complex)) (print 'sorry) nil)
#<STANDARD-METHOD COMBO2 :AROUND (COMPLEX)>
? (defmethod combo2 :around ((x number))
    (print 'around-number-before-call-next-method)
    (print (call-next-method))
    (print 'around-number-after-call-next-method)
    99)
; Try it
? (combo2 17)

AROUND-NUMBER-BEFORE-CALL-NEXT-METHOD
```

```
BEFORE-INTEGER
BEFORE-RATIONAL
PRIMARY
AFTER-RATIONAL
AFTER-INTEGER
1
AROUND-NUMBER-AFTER-CALL-NEXT-METHOD
99
? (combo2 4/5)

AROUND-NUMBER-BEFORE-CALL-NEXT-METHOD
BEFORE-RATIONAL
PRIMARY
AFTER-RATIONAL
1
AROUND-NUMBER-AFTER-CALL-NEXT-METHOD
99
? (combo2 82.3)

AROUND-FLOAT-BEFORE-CALL-NEXT-METHOD
AROUND-NUMBER-BEFORE-CALL-NEXT-METHOD
PRIMARY
1
AROUND-NUMBER-AFTER-CALL-NEXT-METHOD
AROUND-FLOAT-AFTER-CALL-NEXT-METHOD
99
? (combo2 #c(1.0 -1.0))

SORRY
NIL
```

One thing you *can't* do is to provide arguments to `CALL-NEXT-METHOD` that would change the applicable methods. In other words, you need to preserve the type of methods even as you change their values. For example, the following redefinition of one of the `COMBO2` example methods won't work:

```
? (defmethod combo2 :around ((x float))
    (call-next-method (floor x))) ; FLOOR returns an integer
#<STANDARD-METHOD COMBO2 :AROUND (FLOAT)>
? (combo2 45.9)
Error: applicable methods changed
```

Nothing is cast in stone; a peek at the metaobject protocol

The metaobject protocol (MOP) is a collection of functions that expose much of the underlying workings of CLOS. We've already seen one function that belongs to the MOP: `CLASS-PRECEDENCE-LIST`. Some MOP functions, like `CLASS-PRECEDENCE-LIST`, give you visibility into the inner workings of CLOS. Here are some examples:

```
Function                    Argument   Returns
--------                    --------   -------
CLASS-DIRECT-SUBCLASSES     a class    a list of the immediate subclasses
CLASS-DIRECT-SUPERCLASSES   a class    a list of the immediate superclasses
CLASS-DIRECT-SLOTS          a class    a list of non-inherited slots
CLASS-DIRECT-METHODS        a class    a list of non-inherited methods
```

Other MOP functions let you change the underlying behavior of CLOS. You can use this capability to extend CLOS -- perhaps to implement a persistent object store -- or to alter the behavior to more closely correspond to a different kind of object system. Such changes are far beyond the scope of this book. Also, you should be aware that the MOP is *not* (yet, as of early 1999) a standard part of CLOS, and there is no definition of the MOP recognized by any national or international standards body.

So far, the defining document for the MOP is The Art of the Metaobject Protocol. Most Lisp vendors provide at least a partial MOP implementation; you should probably start with the vendor's documentation if you're interested in the MOP.

Contents | Cover
Chapter 13 | Chapter 14 | Chapter 15

Copyright © 1995-2001, David B. Lamkins
All Rights Reserved Worldwide

This book may not be reproduced without the written consent of its author. Online distribution is restricted to the author's site.

Chapter 15 - Closures

In this chapter we'll expand upon the discussion of closures that we started in Chapter 11. We'll see again how (and why) closures capture free variables for use in other execution contexts, then we'll see some practical applications. We'll close this chapter with a look at functions that return functions.

Is it a function of the lifetime, or the lifetime of a function?

Common Lisp does not expose closures per se. Recall from Chapter 11 that a closure is a collection of closed-over variables retained by a function. (A closed-over variable is a variable found "free" in the function; this gets "captured" by the closure. We saw some examples of this in Chapter 11; we'll review the details in the next section, in case you've forgotten.) For this reason, Lisp programmers tend to refer to "a function having closed-over variables" as simply "a closure." Or maybe they call it that because it saves them nine syllables.

A closure has to be associated with a function, so it must have the same lifetime -- or extent -- as the function. But all of the closed-over variables come along for the ride -- a closed-over variable has the same extent as the closure. This means that you can close over a lexical variable, which would normally have lexical extent, and give that variable indefinite extent. This is a very useful technique, as we'll see shortly.

How to spot a free variable, and what to do about it.

A variable is *free* within a function (or within any form, for that matter) if there is no binding occurrence of its name within the lexical scope -- the textual bounds, more or less -- of the function. A binding occurrence is an occurrence of the name that (according to the definition of the form that includes the name) associates storage with the name.

A free variable must be found in one of two places. Either the function is textually wrapped within a form that provides a binding occurrence of the variable, or the variable is `special` (review Chapter 8) and contained in the global environment. If a free variable is not found in one of these two places, it is *unbound* (i.e. has no storage associated with the name) and will cause an error when referenced at runtime.

Using closures to keep private, secure information.

If you close over a lexical variable, that variable is accessible *only* from within the closure. You can use this to your advantage to store information that is truly private, accessible only to functions that have a closure containing your private variable(s).

```
? (let ((password nil)
        (secret nil))
    (defun set-password (new-passwd)
      (if password
        '|Can't - already set|
        (setq password new-passwd)))
    (defun change-password (old-passwd new-passwd)
      (if (eq old-passwd password)
        (setq password new-passwd)
        '|Not changed|))
    (defun set-secret (passwd new-secret)
      (if (eq passwd password)
        (setq secret new-secret)
        '|Wrong password|))
    (defun get-secret (passwd)
      (if (eq passwd password)
        secret
        '|Sorry|)))
GET-SECRET
? (get-secret 'sesame)
|Sorry|
? (set-password 'valentine)
SECRET
? (set-secret 'sesame 'my-secret)
|Wrong password|
? (set-secret 'valentine 'my-secret)
MY-SECRET
? (get-secret 'fubar)
|Sorry|
? (get-secret 'valentine)
MY-SECRET
? (change-password 'fubar 'new-password)
|Not changed|
? (change-password 'valentine 'new-password)
NEW-PASSWORD
? (get-secret 'valentine)
|Sorry|
; The closed-over lexical variables aren't in the global environment
? password
Error: unbound variable
? secret
Error: unbound variable
; The global environment doesn't affect the closed-over variables
? (setq password 'cheat)
CHEAT
? (get-secret 'cheat)
|Sorry|
```

Functions that return functions, and how they differ from macros.

The preceding example is only good for keeping one secret, because every time we evaluate the outer `LET` form we redefine all of the functions that close over our "private" variables. If we want to eliminate our dependence upon the global namespace for functions to manipulate our closed-over variables, we're going to have to find a way to create new closed-over variables and return a function that we can save and later use to manipulate the variables. Something like this will work:

```
? (defun make-secret-keeper ()
    (let ((password nil)
          (secret nil))
      #'(lambda (operation &rest arguments)
          (ecase operation
            (set-password
             (let ((new-passwd (first arguments)))
               (if password
                 '|Can't - already set|
                 (setq password new-passwd))))
            (change-password
             (let ((old-passwd (first arguments))
                   (new-passwd (second arguments)))
               (if (eq old-passwd password)
                 (setq password new-passwd)
                 '|Not changed|)))
            (set-secret
             (let ((passwd (first arguments))
                   (new-secret (second arguments)))
               (if (eq passwd password)
                 (setq secret new-secret)
                 '|Wrong password|)))
            (get-secret
             (let ((passwd (first arguments)))
               (if (eq passwd password)
                 secret
                 '|Sorry|)))))))
MAKE-SECRET-KEEPER
? (defparameter secret-1 (make-secret-keeper))
SECRET-1
? secret-1
#<LEXICAL-CLOSURE #x36AE056>
? (funcall secret-1 'set-password 'valentine)
VALENTINE
? (funcall secret-1 'set-secret 'valentine 'deep-dark)
DEEP-DARK
? (defparameter secret-2 (make-secret-keeper))
SECRET-2
? (funcall secret-2 'set-password 'bloody)
BLOODY
? (funcall secret-2 'set-secret 'bloody 'mysterious)
MYSTERIOUS
? (funcall secret-2 'get-secret 'valentine)
|Wrong password|
? (funcall secret-1 'get-secret 'valentine)
DEEP-DARK
```

The `ECASE` form is an *exhaustive case* statement. In our program, the `OPERATION` must be found in one of the `ECASE` clauses, or Lisp will signal an error.

The `#'(LAMBDA ...` form creates a closure over the free variables `PASSWORD` and `SECRET`. Each time we evaluate `MAKE-SECRET-KEEPER`, the outermost `LET` form creates new bindings for these variables; the closure is then created and returned as the result of the `MAKE-SECRET-KEEPER` function.

> In pre-ANSI Common Lisp, `LAMBDA` is merely a symbol that is recognized as a marker to define a lambda expression. By itself, `LAMBDA` does not create a closure; that is the function of the `#'` reader macro (which expands into a `(FUNCTION ...` form).
>
> ANSI Common Lisp defines a `LAMBDA` macro that expands into `(FUNCTION (LAMBDA ...`, which you can use in place of `#'(LAMBDA` wherever it appears in this example. For backward compatibility with pre-ANSI Common Lisp implementations, you should always write `#'(LAMBDA ...` -- the redundant `(FUNCTION ...` in the expansion will do no harm.

Within each `ECASE` clause we extract arguments from the `&REST` variable `ARGUMENTS` and then do exactly the same processing as in our earlier example.

Once we have invoked `MAKE-SECRET-KEEPER` and saved the resultant closure, we can `FUNCALL` the closure, passing the operation symbol and any additional arguments. Note that each closure created by `MAKE-SECRET-KEEPER` is completely independent; we've therefore achieved the goal of being able to keep multiple secrets.

Functions that return closures are different from macros. A macro is a function that produces a form; the form is then evaluated to produce a result. A function that returns a closure simply returns an object: the closure. The returned closure is *not* automatically evaluated by the Lisp evaluator.

Contents | Cover
Chapter 14 | Chapter 15 | Chapter 16

Copyright © 1995-2001, David B. Lamkins
All Rights Reserved Worldwide

This book may not be reproduced without the written consent of its author. Online distribution is restricted to the author's site.

Chapter 16 - How to Find Your Way Around, Part 2

It's once again time to take a break and learn about some more of the tools you can use to grok [1] the inner workings of Lisp and your programs. In this chapter, we'll learn what the Lisp compiler does to your code, and how to watch what your code does as it runs.

DISASSEMBLE: I always wondered what they put inside those things...

If you understand a little about compilers and assembly language -- or if you're just interminably curious -- you can find out exactly *how* the Lisp compiler translates your Lisp code. `DISASSEMBLE` takes a function name or object and lists the assembly-language instructions that would have been emitted by the Lisp compiler if it actually emitted assembly-language code -- most compilers directly generate machine instructions without invoking an assembler.

The output of `DISASSEMBLE` is dependent both upon the instruction set architecture of the machine you're using to run Lisp and upon the Lisp implementation itself. Here's an example of using `DISASSEMBLE` on a very simple function; this was done using Macintosh Common Lisp on a PowerPC processor.

```
? (defun add1 (n) (1+ n))
ADD1
? (disassemble 'add1)
  (TWNEI NARGS 4)
  (MFLR LOC-PC)
  (BLA .SPSAVECONTEXTVSP)
  (VPUSH ARG_Z)
  (LWZ NARGS 331 RNIL)
  (TWGTI NARGS 0)
  (LI ARG_Y '1)
  (LWZ ARG_Z 0 VSP)
  (BLA .SPRESTORECONTEXT)
  (MTLR LOC-PC)
  (BA .SPBUILTIN-PLUS)
```

The first thing you'll note about this listing is that it looks "Lisp-ish" with the parentheses. The second thing you'll notice -- if you are familiar with the PowerPC instruction set -- is that most of these forms are familiar; it's as if someone took part of a real PowerPC assembly language program and bracketed each line of text in parentheses. You may also notice that there are no comments in the assembly code, that there are some pseudo-instructions such as `VPUSH`, and that this is not a complete program that you could feed into an assembler (even after you stripped off the parentheses). I'll explain all of these points.

Many Lisp systems include an assembler that accepts statements in the form generated by `DISASSEMBLE`. These statements are often named LAP, for Lisp Assembly Program. With the proper documentation, you can write LAP code and have it invoked by your own functions. But you do need the vendor's documentation for this; you can't just find the LAP assembler and feed it a list of LAP instructions. You need to know how to use reserved registers, what subroutines to call, what stack protocol to follow, and many other low-level details. You also need to associate the code with a function name so you can call it later; this is one of the pieces that is missing from the output of `DISASSEMBLE`.

Some Lisp systems provide additional information (beyond raw assembler instructions) in their `DISASSEMBLE` output. In the code above, you'll see that certain reserved registers or memory locations are identified by a distinguishing name, such as `NARGS`, `LOC-PC`, `ARG_Y`, `ARG_Z`, `VSP` and `RNIL`. Sometimes certain instructions (or even short instruction sequences) will be given a mnemonic name that reflects their use by the Lisp compiler; `VPUSH` is one such mnemonic used by this Lisp system.

Some Lisp systems are better than others at including explanatory comments with the disassembled code. Systems that do include comments typically synthesize comments to explain the code, or save information that allows `DISASSEMBLE` to intersperse source program fragments within the disassembly listing.

One useful thing you can do with `DISASSEMBLE` is to see whether *declarations* have any effect on your compiler. Declarations are forms that provide *advice* to the compiler. With the one exception of the `SPECIAL` declaration, which alters the meaning of code that uses it (see Chapter 8) a compiler may or may not use the information that you provide in a declaration. Your Lisp vendor's documentation may provide some guidance as to the effect of declarations, but the best (and most accurate) assessment is made by reading the listing that `DISASSEMBLE` generates.

The previous disassembly of `ADD1` shows that it calls several subroutines: `.SPSAVECONTEXTVSP`, `.SPRESTORECONTEXT`, and `.SPBUILTIN-PLUS`. If that seems like an awful lot of work just to add one to a number, consider that (1) the number can be of any type (including bignum, which is an "infinite" precision integer type), (2) non-numeric arguments are handled gracefully -- you'll get a break into the Lisp debugger rather than a crash or a core dump, and (3) the function probably makes an extra effort to make its presence known to the Lisp debugger.

So, what if we *want* to play fast and loose, assume that `ADD1` will only be called for small integer arguments, and stoically suffer the ungraceful consequences if we screw up and pass the wrong type of data? We can add declarations to express our intent, and then use `DISASSEMBLE` again to see whether the compiler paid any attention to our wishes.

```
? (defun int-add1 (n)
    (declare (fixnum n)
             (optimize (speed 3) (safety 0) (debug 0)))
    (the fixnum (1+ n)))
INT-ADD1
? (disassemble 'int-add1)
  (MFLR LOC-PC)
  (STWU SP -16 SP)
  (STW FN 4 SP)
  (STW LOC-PC 8 SP)
  (STW VSP 12 SP)
  (MR FN TEMP2)
  (LWZ IMM0 -117 RNIL)
  (TWLLT SP IMM0)
  (VPUSH ARG_Z)
  (LWZ ARG_Z 0 VSP)
  (ADDI ARG_Z ARG_Z 4)
  (LWZ LOC-PC 8 SP)
  (MTLR LOC-PC)
  (LWZ VSP 12 SP)
  (LWZ FN 4 SP)
  (LA SP 16 SP)
  (BLR)
```

The `DECLARE` form in `INT-ADD1` includes two kinds of advice. `(FIXNUM N)` declares that the function parameter `N` is a small integer. (The range depends upon your Lisp implementation, but you'll typically get 29-bit fixnums on a 32-bit processor; the remaining three bits are often used by the Lisp system to encode type information.) The `(OPTIMIZE ...` declaration is advice to the compiler that you'd like it to emphasize certain properties of the compiled code. Here, I've said that speed is of ultimate importance, and that I could care less about runtime safety or debuggability. If the compiler pays attention to all of this, I should get code that is optimized for fixnums, runs fast, and falls over if I pass it anything other than a fixnum or cause it to generate a result that isn't a fixnum.

Looking at the generated code, it appears that the compiler *has* paid attention to my declarations. The compiled code for `INT-ADD1` is a bit longer than the code for `ADD1`, but there are *no* subroutine calls. Every instruction generated for `INT-ADD1` is a simple PowerPC instruction (even the `VPUSH` instruction, which is just an alias for a single PowerPC instruction). The addition is performed by PowerPC instructions instead of a subroutine. In fact, most of the code in `INT-ADD1` has to do with entering and leaving the function.

By the way, *some* optimization setting is always in effect if you don't use an `(OPTIMIZE ...` declaration. To find out what are the global optimization settings, do this:

```
? (declaration-information 'optimize)
((SPEED 1) (SAFETY 1) (COMPILATION-SPEED 1) (SPACE 1) (DEBUG 1))
```

> `DECLARATION-INFORMATION` may not exist in a pre-ANSI Common Lisp implementation, but there may be an alternative way to access this information. Consult the vendor's documentation. If that fails, see whether `APROPOS` (see Chapter 10) turns up anything that might be useful.

BREAK and backtrace: How did I end up here?

If you ever need to figure out *exactly* what's going on at a particular point in your program, you can insert a `BREAK` form at the point of interest; when your program evaluates the `BREAK`, the Lisp system will immediately stop your program (without losing any information), and transfer control to the Lisp debugger. Once in the debugger, you can do things like examine the call stack (sometimes named a backtrace, since the stack frames are a trace of your program's current call history, backward in time) and look at local variables at any level in the stack. And, of course, you can execute any Lisp code that you like. But wait, there's more! You can exit the debugger, and your program will continue from where the `BREAK` interrupted it. Or you can change the values of some variables before you continue. If you want, you can provide a value to be returned by the interrupted function. You can even redefine and restart functions anywhere in the call stack.

The fact that `BREAK` is just a Lisp form has its advantages. You can wrap it in a conditional expression of arbitrary complexity, so that your program will trigger the break exactly when it's needed; this is especially useful in debugging loops or recursive functions.

If you have more than one `BREAK` statement in your code, you may find it useful to identify the particular `BREAK` that invokes the debugger. You can provide a format control string and arguments that `BREAK` will use to print a message upon entry to the debugger. The control string and arguments are the same as you'd use for `FORMAT`. (We've seen examples of `FORMAT` in Chapters 4, 5, and 6. Chapter 24 give a a more complete treatment of `FORMAT`.)

> The downside? Most Lisp IDE's don't give you a point-and-click interface to set `BREAK`s. (That *is* a downside, right?)

Lisp defines `BREAK`, the interface for your program to gain entry into the debugger. Once there, the commands that you'll use to navigate are *entirely* implementation-specific. If you're lucky, you'll get a window-and-menus interface to at least the most common actions. If, instead of a GUI, the debugger presents you with just get a message and a prompt, you may have to crack open the manual. But before you get so desperate, try to get the debugger to print a help text or menu: one of the commands `H`, `?`, `:H`, or `:HELP` may work for your Lisp system.

TRACE and STEP: I'm watching you!

When you need to know exactly *how* a function is working at a particular point in your code, `BREAK` and the Lisp debugger are indispensable tools. But they are labor intensive and slow (at least relative to the program's normal execution) -- nothing happens except when you issue commands to the debugger.

Sometimes, it's enough to know that a particular function has been called and returned a value. `TRACE` gives you this ability. You simply invoke trace with one or more function names, and the Lisp environment arranges to print the name of the function and its arguments upon entry, and the name of the function and its values upon exit. All this happens without changing the source code for the function.

```
? (defun factorial (n)
    (if (plusp n)
      (* n (factorial (1- n)))
      1))
FACTORIAL
? (factorial 6)
720
? (trace factorial)
NIL
? (factorial 6)
 Calling (FACTORIAL 6)
  Calling (FACTORIAL 5)
   Calling (FACTORIAL 4)
    Calling (FACTORIAL 3)
     Calling (FACTORIAL 2)
      Calling (FACTORIAL 1)
       Calling (FACTORIAL 0)
       FACTORIAL returned 1
      FACTORIAL returned 1
     FACTORIAL returned 2
    FACTORIAL returned 6
   FACTORIAL returned 24
  FACTORIAL returned 120
 FACTORIAL returned 720
720
```

> Some Lisp systems may print only the first and last lines of this trace, because of compiler optimizations. If you want to see recursive calls, it may help to evaluate `(DECLAIM (OPTIMIZE (DEBUG 3)))` before defining any functions to be traced.

Notice how indentation is used to represent call stack depth. This, and other details of the `TRACE` presentation, are implementation dependent.

When you no longer want to trace a function, evaluate `UNTRACE`, passing the function name (or names). `UNTRACE` without any arguments will stop tracing of all currently traced functions.

Sometimes, despite your best efforts, you're just not sure what parts of a function are being executed. If you're this confused, and you'd rather forge ahead than try to simplify the function, Lisp gives you the `STEP` form. `STEP` takes a complete Lisp form as an argument; it evaluates the form and returns what the form returns. Along the way, though, it lets you see all of the evaluations that happen -- step by step, as it were. Like

`BREAK`, `STEP` only has a standard program interface; the user interface is implementation dependent.

> The quality of information available through `STEP` varies widely among implementations. The most common shortcoming is that you see some transformed version of the program source, rather than the original source code. Generally, you'll be able to spot enough clues (variable names, functions, etc.) so that you can keep your bearings as you execute the stepped code one form at a time.

Footnotes:

[1] :grok: /grok/, var. /grohk/ /vt./ [from the novel "Stranger in a Strange Land", by Robert A. Heinlein, where it is a Martian word meaning literally `to drink' and metaphorically `to be one with'] The emphatic form is `grok in fullness'. 1. To understand, usually in a global sense. Connotes intimate and exhaustive knowledge. Contrast {zen}, which is similar supernal understanding experienced as a single brief flash. 2. Used of programs, may connote merely sufficient understanding.

Contents | Cover
Chapter 15 | Chapter 16 | Chapter 17

Copyright © 1995-2001, David B. Lamkins
All Rights Reserved Worldwide

This book may not be reproduced without the written consent of its author. Online distribution is restricted to the author's site.

Chapter 17 - Not All Comparisons are Equal

Up to this point, I've shown you various comparison functions without really saying much about the differences between them. In this chapter, I'll (finally) tell you about how and why the comparison functions differ and offer some guidelines for their proper use.

The longer the test, the more it tells you

Lisp has a core set of comparison functions that work on virtually any kind of object. These are:

- `EQ`
- `EQL`
- `EQUAL`
- `EQUALP`

The tests with the shorter names support stricter definitions of equality. The tests with the longer implement less restrictive, perhaps more intuitive, definitions of equality. We'll learn about each of the four definitions in the following sections.

EQ is true for identical symbols

`EQ` is true for identical symbols. In fact, it's true for any identical object. In other words, an object is `EQ` to itself. Even a composite object, such as a list, is `EQ` to itself. (But two lists are *not* `EQ` just because they look the same when printed; they must truly be the *same* list to be `EQ`.) Under the covers, `EQ` just compares the memory addresses of objects.

The reason that symbols are *EQ* when they have the same name (and are in the same package) is that the Lisp reader *interns* symbols as it reads them. The first time the reader sees a symbol, it creates it. On subsequent appearances, the reader simply uses the existing symbol.

`EQ` is not guaranteed to be true for identical characters or numbers. This is because most Lisp systems don't assign a unique memory address to a particular number or character; numbers and characters are generally created as needed and stored temporarily in the hardware registers of the processor.

EQL is also true for identical numbers and characters

`EQL` retains `EQ`'s notion of equality, and extends it to identical numbers and characters. Numbers must agree in value *and* type; thus 0.0 is *not* `EQL` to 0. Characters must be truly identical; `EQL` is case sensitive.

EQUAL is usually true for things that print the same

`EQ` and `EQL` are not generally true for lists that print the same. Lists that are not `EQ` but have the same structure will be indistinguishable when printed; they will also be `EQUAL`.

Strings are also considered `EQUAL` if they print the same. Like `EQL`, the comparison of characters within strings is case-sensitive.

EQUALP ignores number type and character case

`EQUALP` is the most permissive of the core comparison functions. Everything that is `EQUAL` is also `EQUALP`. But `EQUALP` ignores case distinctions between characters, and applies the (typeless) mathematical concept of equality to numbers; thus 0.0 is `EQUALP` to 0.

Furthermore, `EQUALP` is true if corresponding elements are `EQUALP` in the following composite data types:

- Arrays
- Structures
- Hash Tables

Longer tests are slower; know what you're comparing

The generality of the above longer-named tests comes with a price. They must test the types of their arguments to decide what kind of equality is applicable; this takes time.

`EQ` is blind to type of an object; either the objects are the same object, or they're not. This kind of test typically compiles into one or two machine instructions and is very fast.

You can avoid unnecessary runtime overhead by using the most restrictive (shortest-named) test that meets your needs.

Specialized tests run faster on more restricted data types

If you know the type of your data in advance, you can use comparisons that are specialized to test that particular type of data. Tests are available for characters, strings, lists, and numbers. And, of course, there are also comparisons for other relationships besides equality.

- Characters
    - `CHAR=`
    - `CHAR/=`
    - `CHAR<`
    - `CHAR<=`
    - `CHAR>`
    - `CHAR>=`
    - `CHAR-EQUAL`
    - `CHAR-NOT-EQUAL`
- Strings
    - `STRING=`
    - `STRING/=`
    - `STRING<`
    - `STRING<=`
    - `STRING>`
    - `STRING>=`
    - `STRING-EQUAL`
    - `STRING-NOT-EQUAL`
- Lists
    - `TREE-EQUAL`

- Numbers
    - =
    - /=
    - <
    - <=
    - >
    - >=

Contents | Cover
Chapter 16 | Chapter 17 | Chapter 18

Copyright © 1995-2001, David B. Lamkins
All Rights Reserved Worldwide

This book may not be reproduced without the written consent of its author. Online distribution is restricted to the author's site.

Chapter 18 - Very Logical, Indeed...

Now it's time to look at things having to do with boolean (true and false) logic. We'll learn about common logical functions, and conditional evaluation. If you're a bit twiddler, this chapter should warm your heart: we'll introduce bit manipulation functions, bit vectors, and generalized byte manipulation.

AND and OR evaluate only as much as they need

`AND` and `OR` are macros in Common Lisp. This means that they have control over when (and *if*) their arguments get evaluated. `AND` and `OR` take advantage of this ability: they stop evaluating their arguments as soon as they determine an answer.

Consider `AND`: it evaluates its arguments, starting with the leftmost, only as long as each argument evaluates to a true (i.e. not `NIL`) value. As soon as `AND` evaluates the leftmost false (`NIL`) argument, its work is done -- the result will be `NIL` no matter how many more true arguments it evaluates, so `AND` just returns `NIL` without evaluating any more of its arguments. (Think of this as a "one strike and you're out" policy.) `AND` returns true only if all of its arguments evaluate to a true value.

In fact, `AND` returns either `NIL` (if one of its arguments evaluates to `NIL`) or the non-`NIL` value of its rightmost argument. Some Lisp programmers take advantage of this to treat `AND` as a simple conditional.

```
? (defun safe-elt (sequence index)
    (and (<-1 index (length sequence)) ; guard condition
         (values (elt sequence index) t)))
SAFE-ELT
? (safe-elt #(1 2 3) 3)
NIL
? (elt #(1 2 3) 3)
Error: index out of bounds
? (safe-elt #(1 2 3) 2)
3
T
```

`OR` also evaluates only enough arguments to determine its result: it evaluates arguments, starting with the leftmost, so long as they evaluate to `NIL`. The first non-`NIL` result is returned as `OR`'s value; arguments further to the right are not evaluated.

One caution is in order about `AND` and `OR`. Because they are macros, and not functions, they can not be used for mapping (see Chapter 12). Use the predicate mapping functions (`SOME`, `EVERY`, etc.) instead.

Bits, bytes, and Boole

Machine languages and low-level programming languages always provide the ability to perform bitwise boolean operations: groups of bits are logically combined on a bit-by-bit basis; adjacent bits have no effect on their neighbors in determining the result. The same languages also let you treat adjacent groupings of bits as a unit; this is commonly called a byte or a bit field. Usually bitwise and bit field operations are constrained by the size of hardware registers.

Lisp makes these same facilities available, but removes the constraints that might otherwise be imposed by the underlying hardware.

Sixteen bitwise boolean operations are available in Lisp through the `BOOLE` function. `BOOLE` is a three-argument functions expecting an operation designator plus two integer arguments and producing an integer result. Remember that Lisp has infinite precision integers (bignums), so these bitwise boolean operations are exempt from machine limitations (except for available memory).

The operation designator is a constant value having a name from the following list. The actual values of these constants is specific to the Lisp implementation.

1. `BOOLE-1` *; returns arg1*
2. `BOOLE-2` *; returns arg2*
3. `BOOLE-ANDC1` *; and complement of arg1 with arg2*
4. `BOOLE-ANDC2` *; and arg1 with complement of arg2*
5. `BOOLE-AND` *; and arg1 with arg2*
6. `BOOLE-C1` *; complement of arg1*
7. `BOOLE-C2` *; complement of arg2*
8. `BOOLE-CLR` *; always all zeroes*
9. `BOOLE-EQV` *; exclusive-nor of arg1 with arg2 (equivalence)*
10. `BOOLE-IOR` *; inclusive-or of arg1 with arg2*
11. `BOOLE-NAND` *; not-and of arg1 with arg2*
12. `BOOLE-NOR` *; not-or of arg1 with arg2*
13. `BOOLE-ORC1` *; or complement of arg1 with arg2*
14. `BOOLE-ORC2` *; or arg1 with complement of arg2*
15. `BOOLE-SET` *; always all ones*
16. `BOOLE-XOR` *; exclusive-or of arg1 with arg2*

```
? (boole boole-and 15 7)
7
? (boole boole-ior 2 3)
3
? (boole boole-set 99 55)
-1
? (boole boole-andc2 7 4)
3
```

There are also eleven bitwise logical functions; these are similiar to the `BOOLE` operations, except that the constant and identity operations are not present in this group, and the complement function takes only one argument. (Except for `LOGNOT`, all of the following functions expect two arguments.)

1. `LOGAND`
2. `LOGANDC1`
3. `LOGANDC2`
4. `LOGEQV`
5. `LOGIOR`
6. `LOGNAND`
7. `LOGNOR`
8. `LOGNOT`
9. `LOGORC1`
10. `LOGORC2`
11. `LOGXOR`

`LOGTEST` returns true if any of the corresponding bits in its two arguments are both ones.

```
? (logtest 7 16)
NIL
? (logtest 15 5)
T
```

`LOGBITP` tests one bit in the two's complement representation of an integer, returning `T` if the bit is 1 and `NIL` if the bit is 0. The least significant (rightmost) bit is bit 0.

```
? (logbitp 0 16)
NIL
? (logbitp 4 16)
T
? (logbitp 0 -2)
NIL
? (logbitp 77 -2)
T
```

`LOGCOUNT` counts 1 bits in the binary representation of a positive integer, and 0 bits in the two's complement binary representation of a negative number.

```
? (logcount 35)
3
? (logcount -2)
1
```

Bit vectors can go on forever

A vector composed of only 1s and 0s has a compact representation as a *bit vector*, a special representation for printing and reading, and a set of logical operations. Like all vectors (and arrays) in Common Lisp, the size of a bit vector is limited by the constant `ARRAY-TOTAL-SIZE-LIMIT`; this can be as small as 1,024, but is typically large enough that the size of memory sets a practical limit on the size of bit-vectors.

The printed representation of a bit vector begins with the `#*` reader macro, followed by 1s and 0s. The bit vector's length is determined by the 1s and 0s that make up its elements. (The printed representation of an empty bit vector is `#*`.)

```
? #*0010101
#*0010101
? (length #*0010101)
7
```

There are eleven bitwise logical operations available for bit vectors. With the exception of `BIT-NOT`, these are all functions of two arguments. Unlike the corresponding bitwise logical operations on integers, the bit vector logical operations expect their arguments to be of the same size.

1. `BIT-AND`
2. `BIT-ANDC1`
3. `BIT-ANDC2`
4. `BIT-EQV`
5. `BIT-IOR`
6. `BIT-NAND`
7. `BIT-NOR`
8. `BIT-NOT`
9. `BIT-ORC1`

10. `BIT-ORC2`
11. `BIT-XOR`

These functions will destructively update a result bit vector if you provide an optional third (second in the case of `BIT-NOT`) argument. If the optional argument is `T`, then the first argument will be updated with the result bits. If the optional argument is a bit vector, it will be updated with the result bits and the input arguments will be unchanged. (This in-place update is not available for bitwise operations on integers; destructive bit vector operations may be more efficient once the number of bits exceeds the size of a fixnum.)

```
? (bit-and #*00110100 #*10101010)
#*00100000
? (bit-ior #*00110100 #*10101010)
#*10111110
? (bit-not #*00110100)
#*11001011
```

You can access an individual element of a bit vector using `BIT`. This is a vector accessor, and not a boolean test, so it returns 0 or 1. `BIT` can also be used in a `SETF` form to alter an element of a bit vector.

```
? (bit #*01001 1)
1
? (let ((bv (copy-seq #*00000)))
    (setf (bit bv 3) 1)
    bv)
#*00010
```

Chunks of bits make bytes

Getting back to integer manipulation as we wrap up this chapter, we'll see how to manipulate fields of adjacent bits within an integer value.

The first thing we need when manipulating a field of bits (called a *byte* in Common Lisp) is a way of specifying its bounds. The `BYTE` function constructs a byte specifier from a size (number of bits) and a position (the number of the rightmost bit of the byte within the containing integer, where the LSB is bit 0). The representation of a byte specifier depends upon the Lisp implementation.

The functions `BYTE-SIZE` and `BYTE-POSITION` extract the size and position values from a byte specifier.

```
? (setq bs (byte 5 3)) ; 5 bits, rightmost has weight 2^3 in source
248 ; implementation-dependent
? (byte-size bs)
5
? (byte-position bs)
3
```

You can extract and replace bytes from an integer using the functions `LDB` (load byte) and `DPB` (deposit byte).

```
? (ldb (byte 8 8) 258)
1
? (ldb (byte 8 0) 258)
2
? (dpb 3 (byte 8 8) 0)
768
? (dpb 1 (byte 1 5) 1)
33
```

`LDB-TEST` returns true if any of the bits are 1 in a specified byte.

```
? (ldb-test (byte 3 2) 3)
NIL
? (ldb-test (byte 3 2) 9)
T
? (ldb-test (byte 3 2) 34)
NIL
```

`INTEGER-LENGTH` tells you how many bits are necessary to represent an integer in two's complement form. A positive integer will always have an unsigned representation using the number of bits determined by `INTEGER-LENGTH`. A negative integer has a signed binary representation that requires one bit more than the number of bits determined by `INTEGER-LENGTH`.

```
? (integer-length 69) ; 1000101
7
? (integer-length 4) ; 100
3
? (integer-length -1) ; 1
0
? (integer-length 0)
0
? (integer-length -5) ; 1011
3
```

You can shift the bits in an integer using the `ASH` function. This is an *arithmetic* shift; it treats the integer as a two's complement binary number and preserves the sign (leftmost) bit as the rest of the bits are shifted. A left shift shifts bits to the left, replacing them with zeroes (and preserving the sign bit). A right shift shifts bits to the right, replacing them with zeroes (and preserving the sign bit).

`ASH` expects two arguments, an integer to be shifted, and a shift count. A shift count of 0 returns the integer unchanged. A positive count shifts bits to the left by the specified number of positions. A negative count shifts bits to the right.

```
? (ash 75 0)
75
? (ash 31 1)
62
? (ash -7 1)
-14
? (ash 32 8)
8192
? (ash -1 8)
-256
? (ash 16 -1)
8
? (ash 11 -1)
5
? (ash 32 -8)
0 ; all one bits shifted out
? (ash -99 -2)
-25
```

Contents | Cover
Chapter 17 | Chapter 18 | Chapter 19

Copyright © 1995-2001, David B. Lamkins
All Rights Reserved Worldwide

This book may not be reproduced without the written consent of its author. Online distribution is restricted to the author's site.

Chapter 19 - Streams

All of the I/O functions in Lisp accept a stream argument. In some cases (e.g. `READ` and `PRINT`) the stream argument is optional; by default, input comes from the `*STANDARD-INPUT*` stream (normally connected to the keyboard) and output goes to the `*STANDARD-OUTPUT*` stream (normally connected to the display). You can redirect I/O by either providing optional stream arguments to `READ` and `PRINT` (as well as other I/O functions), or by binding `*STANDARD-INPUT*` and `*STANDARD-OUTPUT*` to different streams. (We'll see both of these approaches used in the following examples.)

Streams provide a pipe to supply or accept data

Throughout the preceding chapters of this book, streams have been involved whenever we've seen an example that does input or output -- and *all* of the examples do I/O, if you count our interactions with the listener. A Lisp stream can provide (source) or consume (sink) a sequence of bytes or characters. (Remember the Lisp definition of byte: a byte can contain any number of bits.)

Some I/O functions accept `T` or `NIL` as a stream designator. `T` is a synonym for `*TERMINAL-IO*`, a bidirectional (input *and* output) stream which conventionally reads from `*STANDARD-INPUT*` and writes to `*STANDARD-OUTPUT*`. `NIL` is a synonym for `*STANDARD-INPUT*` when used in a context which expects an input stream, or for `*STANDARD-OUTPUT*` when used in a context which expects an output stream.

> `FORMAT` (which we've already seen in several examples, and will examine in depth in Chapter 24) expects as its first argument a stream, a `T`, a `NIL`, or a string with a fill pointer. In this case, however, the `NIL` designator causes `FORMAT` to return a string, rather than write to `*STANDARD-OUTPUT*` as is the case for other I/O functions.

The power of streams comes from the ability to associate a stream with a file, a device (such as keyboard, display, or network), or a memory buffer. Program I/O can be directed at will by simply creating the appropriate type of stream for your program to use. The I/O implementation is abstracted away by the stream so your program won't have to be concerned with low-level details.

Lisp also provides a number of special-purpose streams which serve to combine or manipulate other streams in novel ways. A `TWO-WAY-STREAM` combines a separate input stream and output stream into an I/O stream. A `BROADCAST-STREAM` sends output to zero or more output streams; think of this as a bit-bucket when used with zero streams, and a broadcaster when used with multiple streams. A `CONCATENATED-STREAM` accepts input requests on behalf of zero or more input streams; when one stream's input is exhausted, the `CONCATENATED-STREAM` begins reading from its next input stream. An `ECHO-STREAM` is like a `TWO-WAY-STREAM`, with the added feature that anything your program reads from the `TWO-WAY-STREAM`'s input stream automatically gets echoed to the corresponding output stream. Finally, a `SYNONYM-STREAM` is an alias for another stream; the alias can be changed at runtime without creating a new `SYNONYM-STREAM`.

Quite a few I/O functions operate directly on streams:

`READ-BYTE` *stream &optional eof-error-p eof-value* and `READ-CHAR` *&optional stream eof-error-p eof-value recursive-p*

> Reads a byte or a character from an input stream.

`WRITE-BYTE` *byte stream* and `WRITE-CHAR` *char &optional stream*

> Writes a byte or a character to an output stream.

`READ-LINE` *&optional stream eof-error-p eof-value recursive-p* and `WRITE-LINE` *string &optional stream &key start end*

Read or write a line of text, terminated by a newline. (The newline is consumed and discarded on input, and added to output.) The `:START` and `:END` keyword arguments let you limit the portion of the string written by `WRITE-LINE`.

`WRITE-STRING` *string &optional stream &key start end*

Like `WRITE-LINE`, but does not append a newline to output.

`PEEK-CHAR` *&optional peek-type stream eof-error-p eof-value recursive-p*

Reads a character from an input stream without consuming the character. (The character remains available for the next input operation.) Optional argument *peek-type* alters `PEEK-CHAR`'s behavior to first skip whitespace (*peek-type* `T`) or to first skip forward to some specified character (*peek-type* a character).

`UNREAD-CHAR` *character &optional stream*

Pushes a character (which must be the character most recently read) back onto the front of an input stream, where it remains until read again.

`LISTEN` *&optional stream*

Returns true if data is available (e.g. a yet-to-be-read keystroke or unconsumed file data) on an input stream.

`READ-CHAR-NO-HANG` *&optional stream eof-error-p eof-value recursive-p*

If a character is available on the input stream, return the character. Otherwise, return `NIL`.

`TERPRI` *&optional stream* and `FRESH-LINE` *&optional stream*

`TERPRI` unconditionally writes a newline to an output stream. `FRESH-LINE` writes a newline unless it can determine that the output stream is already at the beginning of a new line; `FRESH-LINE` returns `T` if it actually wrote a newline, and `NIL` otherwise.

`CLEAR-INPUT` *&optional stream*

Flushes unread data from an input stream, if it makes sense to do so.

`FINISH-OUTPUT` *&optional stream*, `FORCE-OUTPUT` *&optional stream*, and `CLEAR-OUTPUT` *&optional stream*

These functions flush output buffers if it makes sense to do so. `FINISH-OUTPUT` tries to make sure that buffered output reaches its destination, then returns. `FORCE-OUTPUT` attempts to initiate output from the buffer, but does not wait for completion like `FINISH-OUTPUT`. `CLEAR-OUTPUT` attempts to discard buffered data and abort any output still in progress.

In the read functions listed above, optional arguments `EOF-ERROR-P` and `EOF-VALUE` specify what happens when your program makes an attempt to read from an exhausted stream. If `EOF-ERROR-P` is true (the default), then Lisp will signal an error upon an attempt to read an exhausted stream. If `EOF-ERROR-P` is `NIL`, then Lisp returns `EOF-VALUE` (default `NIL`) instead of signalling an error.

Optional argument `RECURSIVE-P` is reserved for use by functions called by the Lisp reader.

Creating streams on files

The `OPEN` function creates a `FILE-STREAM`. Keyword arguments determine attributes of the stream (`:DIRECTION`, `:ELEMENT-TYPE`, and `:EXTERNAL-FORMAT`) and how to handle exceptional conditions (`:IF-EXISTS` and `:IF-DOES-NOT-EXIST`). If `OPEN` is successful it returns a stream, otherwise it returns `NIL` or signals an error.

Keyword	Value	Stream Direction
:DIRECTION	:INPUT	input (default)
:DIRECTION	:OUTPUT	output

:DIRECTION	:IO	input & output
:DIRECTION	:PROBE	none, returns a closed stream

Keyword	Value	Action if File Exists
:IF-EXISTS	NIL	return NIL
:IF-EXISTS	:ERROR	signal an error
:IF-EXISTS	:NEW-VERSION	next version (or error)
:IF-EXISTS	:RENAME	rename existing, create new
:IF-EXISTS	:SUPERSEDE	replace file upon CLOSE
:IF-EXISTS	:RENAME-AND-DELETE	rename and delete existing, create new
:IF-EXISTS	:OVERWRITE	reuse existing file (position at start)
:IF-EXISTS	:APPEND	reuse existing file (position at end)

Keyword	Value	Action if File Does Not Exist
:IF-DOES-NOT-EXIST	NIL	return NIL
:IF-DOES-NOT-EXIST	:ERROR	signal an error
:IF-DOES-NOT-EXIST	:CREATE	create the file

Keyword	Value	Element Type
:ELEMENT-TYPE	:DEFAULT	character (default)
:ELEMENT-TYPE	'CHARACTER	character
:ELEMENT-TYPE	'SIGNED-BYTE	signed byte
:ELEMENT-TYPE	'UNSIGNED-BYTE	unsigned byte
:ELEMENT-TYPE	*character subtype*	character subtype
:ELEMENT-TYPE	*integer subtype*	integer subtype
:ELEMENT-TYPE	*other*	implementation-dependent

Keyword	Value	File Format
:EXTERNAL-FORMAT	:DEFAULT	default (default)
:EXTERNAL-FORMAT	*other*	implementation-dependent

Once you've opened a stream, you can use it with appropriate input or output functions, or with queries that return attributes of either the stream or the file. The following queries can be applied to all kinds of streams. All of these accept a stream argument:

Function	Returns
INPUT-STREAM-P	true if stream can provide input
OUTPUT-STREAM-P	true if stream can accept output
OPEN-STREAM-P	true if stream is open
STREAM-ELEMENT-TYPE	the type specifier for stream elements
INTERACTIVE-STREAM-P	true if stream is interactive (e.g. keyboard/display)

These queries can be applied to file streams. These also accept a stream argument:

Function	Returns
STREAM-EXTERNAL-FORMAT	implementation-dependent
FILE-POSITION	current file offset for read or write, or NIL
FILE-LENGTH	length of stream, or NIL

`FILE-POSITION` returns a byte offset within the stream. This is an exact count for streams of integer subtypes (see below for further description of binary I/O). For streams of character subtypes, the position is guaranteed only to increase during reading or writing; this allows for variations in text record formats and line terminators.

`FILE-POSITION` can also be called with a second argument to change the file offset for the next read or write. When used for this purpose, `FILE-POSITION` returns true when it succeeds.

You should always close a stream when you're done using it (except for the interactive streams provided for you use by Lisp, such as `*STANDARD-INPUT*`, `*STANDARD-OUTPUT*`, and `*TERMINAL-IO*`). The "open, process, close" pattern is very common, so Lisp provides macros to make the pattern both easy to code and error-free.

`WITH-OPEN-FILE` is tailored for file streams. Its arguments are a variable to be bound to the stream, a pathname, and (optionally) keyword arguments suitable for `OPEN`. The stream is always closed when control leaves the `WITH-OPEN-FILE` form.

```
(with-open-file (stream "my-file.dat" :direction :input)
  ... do something using stream ...)
```

`WITH-OPEN-STREAM` expects a variable name and a form to be evaluated; the form should produce a stream value or `NIL`. This macro is commonly used with constructors for specialty streams, such as `MAKE-BROADCAST-STREAM`, `MAKE-ECHO-STREAM`, `MAKE-TWO-WAY-STREAM`, `MAKE-CONCATENATED-STREAM`, and `MAKE-SYNONYM-STREAM`.

Creating streams on strings

The data read or written by a stream doesn't have to be associated with a device -- the data can just as well be in memory. String streams let you read and write at memory speeds, but they can't provide either file or interactive capabilities. Lisp provides constructors (`MAKE-STRING-INPUT-STREAM` and `MAKE-STRING-OUTPUT-STREAM`), plus macros to support the "open, process, close" pattern.

```
? (with-input-from-string (stream "This is my input via stream.")
    (read stream))
THIS
? (with-output-to-string (stream)
    (princ "I'm writing to memory!" stream))
"I'm writing to memory!"
```

These macros accept keyword and optional arguments. `WITH-INPUT-FROM-STRING` allows `:BEGIN` and `:END` keyword arguments to establish bounds on the portion of the string read via the stream. A `:INDEX` keyword argument lets you name a variable to receive the offset of the next string element to be read -- this is set only upon leaving the `WITH-INPUT-FROM-STRING` form.

`WITH-OUTPUT-TO-STRING` allows an optional form, which is evaluated to produce the output string; if this form is missing or `NIL`, the macro creates a string for you using the `:ELEMENT-TYPE` keyword argument.

Binary I/O

Lisp supports binary I/O via streams whose element types are finite (i.e. bounded) subtypes of `INTEGER`. Some examples of appropriate types are:

- Implementation-dependent

    - `SIGNED-BYTE`
    - `UNSIGNED-BYTE`
- Range of values
    - `(INTEGER 0 31)`
    - `(INTEGER -16 15)`
- Specific number of bits
    - `(SIGNED-BYTE 8)`
    - `(UNSIGNED-BYTE 6)`
    - `BIT`

ANSI Common Lisp implementations should support any of these types for binary I/O. However, the implementation is not required to directly map the specified `:ELEMENT-TYPE` onto the underlying file system; an implementation is permitted to alter the external format so long as data read from a binary file is the same as that written using the same `:ELEMENT-TYPE`.

Contents | Cover
Chapter 18 | Chapter 19 | Chapter 20

Copyright © 1995-2001, David B. Lamkins
All Rights Reserved Worldwide

This book may not be reproduced without the written consent of its author. Online distribution is restricted to the author's site.

Chapter 20 - Macro Etiquette

Macros in Lisp are much more capable than macros in other programming languages. Rather than just providing a simple shorthand notation, Lisp macros give you the capability to truly extend the language. In this chapter we'll learn about the program transforming capabilities of macros as we see how to properly construct macros to solve a wide variety of problems.

Macros are programs that generate programs

Mention macros to most programmers, perhaps even you, and the first image that comes to mind is string substitution -- you use a macro to glue together a few parameters in a new way, maybe with a bit of compile-time decision making thrown in. And because of the typical (in languages other than Lisp) disparity between the macro language and the programming language, the difficulty of writing a macro increases much faster than its complexity.

Lisp macros are Lisp programs that generate other Lisp programs. The generated Lisp code has a fully-parenthesized notation, as does the macro that generates the code. In the simplest case, a macro substitutes forms within a template, clearly establishing a visual correspondence between the generating code and the generated code. Complex macros can use the full power of the Lisp language to generate code according to the macro parameters; often a template form is wrapped in code that constructs appropriate subforms, but even this approach is just a typical use pattern and not a requirement (or restriction) of the Lisp macro facility.

In the following sections, we'll examine the mechanism by which the Lisp system translates code generated by a macro, then we'll see several increasingly sophisticated examples of macros. We'll finish this chapter with a comparison of macros versus the use of inline function declarations.

Close up: how macros work

You define a macro with a `DEFMACRO` form, like this:

```
(defmacro name (arguments ...)
  body)
```

`DEFMACRO` is like `DEFUN`, but instead of returning values, the body of the `DEFMACRO` returns a Lisp form. (As we'll see shortly, there's a very simple way to generate this form with selected subforms replaced by parameters from the macro call or computed by the macro's program.)

Your program "calls" a macro the same way it calls a function, but the behavior is quite different. First, none of the macro's parameters are evaluated. Ever. Macro parameters are bound literally to the corresponding arguments in the macro definition. If you pass `(* 7 (+ 3 2))` to a macro, the argument in the body of the macro definition is bound to the literal list `(* 7 (+ 3 2))`, and *not* the value 35.

Next, the *macro expander* is invoked, receiving all of the actual parameters bound to their corresponding arguments as named by the `DEFMACRO` form. The macro expander is just the body of the `DEFMACRO` form, which is just Lisp code; the only catch is that the Lisp system expects the macro expander to return a Lisp form.

The Lisp system then evaluates whatever form the macro expander returns. If the returned form is a macro, it gets expanded. Otherwise, the form is evaluated by the rules we first learned in Chapter 3, Lesson 2.

> The preceding paragraph is conceptually correct. However, a Lisp implementation may expand macros at different times. A macro could be expanded just once, when your program is compiled. Or it could be expanded on first use as your program runs, and the expansion could be cached for subsequent reuse. Or the macro could be expanded every time it's used. A properly written macro will behave the same under all of these implementations.

In Chapter 21 we'll expand upon some of the things you can express with argument lists.

Backquote looks like a substitution template

The simplest way to generate a form in the body of your macro expander is to use the *backquote* (`` ` ``) reader macro. This behaves like the *quote* (`'`) reader macro, except for when a comma (`,`) appears in the backquoted form.

> A comma is only permitted in a backquoted form. If you use a comma in a quoted form, Lisp will signal an error when it reads the form.

Like quote, backquote suppresses evaluation. But a comma within a backquoted form "unsuppresses" evaluation for just the following subform.

```
? `(The sum of 17 and 83 is ,(+ 17 83))
(THE SUM OF 17 AND 83 IS 100)
```

Compare the preceding example, which used backquote, with the similar form using quote (and omitting the comma).

```
? '(The sum of 17 and 83 is (+ 17 83))
(THE SUM OF 17 AND 83 IS (+ 17 83))
```

You can probably imagine how backquote and comma provide a template with substitution capabilities. This is just what we need for our macro expander. Here are a couple of simple examples.

```
; Define the macro
? (defmacro swap (a b) ; NOTE: This is a restricted version of ROTATEF
    `(let ((temp ,a))
        (setf ,a ,b)
        (setf ,b temp)))
SWAP

; First invocation
? (let ((x 3)
         (y 7))
    (swap x y) ; macro call
    (list x y))
(7 3)
; Let's see the form generated by SWAP:
? (pprint (macroexpand-1 '(swap x y)))

(LET ((TEMP X))
  (SETF X Y)
  (SETF Y TEMP))

; Second invocation
```

```
? (let ((c (cons 2 9))) ; (2 . 9)
    (swap (car c) (cdr c))
    c)
(9 . 2)
; And the expansion of its macro call
? (pprint (macroexpand-1 '(swap (car c) (cdr c))))

(LET ((TEMP (CAR C)))
  (SETF (CAR C) (CDR C))
  (SETF (CDR C) TEMP))

; Here's the second invocation again, "macroexpanded" by hand.
? (let ((c (cons 2 9)))
    (LET ((TEMP (CAR C)))
      (SETF (CAR C) (CDR C))
      (SETF (CDR C) TEMP))
    c)
(9 . 2)
```

`(PPRINT (MACROEXPAND-1 '`*`macro-call`*`))` is a very handy tool to see what form your macro expander generates. (Don't worry if the output from your Lisp system looks exactly as shown here; there may be some differences in layout.)

As you look at these examples, the important things to note are that:

1. the macro arguments receive the literal representation of their actual parameters from the macro call, and
2. macro arguments that are preceded by a comma within a backquoted form are substituted with the literal representation of the parameter from the macro call.

Here are some more macro definitions. Experiment with these in your Lisp system to see what they do.

```
(defmacro sortf (place)
  `(setf ,place (sort ,place)))

(defmacro togglef (place)
  `(setf ,place (not ,place)))

(defmacro either (form1 form2)
  ; (random 2) returns 0 or 1
  `(if (zerop (random 2)) ,form1 ,form2))
```

Beyond the obvious, part 1: compute, then generate

Macros start to get interesting when they do more than a simple textual substitution. In this section, we'll explore a real-world example of using a macro to extend Lisp into the problem domain. In addition to providing a macro expander, our new macro will automatically generate an environment that will be referenced by the expander. Our example will show how to move computations from run-time to compile-time, and how to share information computed at compile-time.

Let's say you're working on an interactive game that makes heavy use of the trigonometric function *sine r* in computing player motion and interaction. You've already determined that calling the Lisp function `SIN` is too time-consuming; you also know that your program will work just fine with approximate results for the computation of *sine r*. You'd like to define a `LOOKUP-SIN` macro to do the table lookup at runtime; you'd also like to hide the details of table generation, an implementation detail with which you'd rather not clutter your program's source code.

Your macro will be invoked as `(LOOKUP-SIN` *radians divisions*`)`, where *radians* is always in the range of zero to one-quarter pi, and *divisions* is the number of discrete values available as the result of `LOOKUP-SIN`. At runtime, the macro expander will just compute the index into a lookup table, and return the value from the table. The table will be generated at compile-time (on most Lisp systems). Furthermore, only one table will ever be generated for a given value of *divisions* in the macro call.

Here's the code. The comments and documentation strings should help you to understand the code as you read it. I'll provide further explanation below.

```
;; This is where we cache all of the sine tables generated
;; during compilation. The tables stay around at runtime
;; so they can be used for lookups.
(defvar *sin-tables* (make-hash-table)
  "A hash table of tables of sine values. The hash is keyed
by the number of entries in each sine table.")

;; This is a helper function for the LOOKUP-SIN macro.
;; It is used only at compile time.
(defun get-sin-table-and-increment (divisions)
  "Returns a sine lookup table and the number of radians quantized
by each entry in the table. Tables of a given size are reused.
A table covers angles from zero to pi/4 radians."
  (let ((table (gethash divisions *sin-tables* :none))
        (increment (/ pi 2 divisions)))
    (when (eq table :none)
      ;; Uncomment the next line to see when a table gets created.
      ;;(print '|Making new table|)
      (setq table
            (setf (gethash divisions *sin-tables*)
                  (make-array (1+ divisions) :initial-element 1.0)))
      (dotimes (i divisions)
        (setf (aref table i)
              (sin (* increment i)))))
    (values table increment)))

;; Macro calls the helper at compile time, and returns an
;; AREF form to do the lookup at runtime.
(defmacro lookup-sin (radians divisions)
  "Return a sine value via table lookup."
  (multiple-value-bind (table increment)
                       (get-sin-table-and-increment divisions)
    `(aref ,table (round ,radians ,increment))))
```

If you still don't see the point of all this code after having read the introduction to this section and the comments in the code, here it is: when your program runs, it executes *just* `AREF` (and associated `ROUND`) to look up the *sin r* value.

```
? (pprint (macroexpand-1 '(lookup-sin (/ pi 4) 50)))

(AREF #(0.0 0.0314107590781283 0.06279051952931337
        [additional entries not shown]
        0.9980267284282716 0.9995065603657316 1.0)
      (ROUND (/ PI 4) 0.031415926535897934))
;; Note that the macro call makes no mention of a lookup table.
;; Tables are generated as-needed by (and for) the compiler.
? (lookup-sin (/ pi 4) 50)
```

```
0.7071067811865476
```

In the macroexpansion, the `#(...)` is the printed representation of the lookup table for 50 divisions of the quarter circle. This table is stored in the `*SIN-TABLES*` hash table, where it is shared by *every* macro call to `(LOOKUP-SIN` *angle* `50)`. We don't even have to do a hash lookup at runtime, because the macro expander has captured the free variable `TABLE` from the `MULTIPLE-VALUE-BIND` form in `LOOKUP-SIN`.

Beyond the obvious, part 2: macros that define macros

Macros that define macros are used infrequently, partly because it's hard to think of a good use for this technique and partly because it's difficult to get right. The following macro, based upon an example in Paul Graham's "On Lisp" book, can be used to define synonyms for the names of Lisp functions, macros, and special forms.

```
? (defmacro defsynonym (old-name new-name)
    "Define OLD-NAME to be equivalent to NEW-NAME when used in
the first position of a Lisp form."
    `(defmacro ,new-name (&rest args)
       `(,',old-name ,@args)))
DEFSYNONYM
? (defsynonym make-pair cons)
MAKE-PAIR
? (make-pair 'a 'b)
(A . B)
```

Macros are always a little bit dangerous because code containing a macro call does not automatically get updated if you change the definition of the macro. You can always establish your own convention to help you remember that you need to recompile certain code after you change a macro definition. But there's always the possibility that you'll forget, or make a mistake.

Ultimately, the likelihood that you'll inadvertently end up with code that was compiled with an old version of a macro is directly proportional to how often you're likely to change the macro. I'll probably never need to change the `LOOKUP-SIN` macro from the previous section once it's defined and working. On the other hand, a macro like `DEFSYNONYM` practically begs to be used again and again as you generate new code. If you change your mind about the *old name* to associate with a given *new name*, all of your previously compiled code will still refer to the `old name` that you had decided upon previously.

```
;; WARNING: This example illustrates a practice to avoid!

;; Here's some core algorithm
? (defun process-blah-using-algorithm-zark (...) ...)
PROCESS-BLAH-USING-ALGORITHM-ZARK

;; And here's where I use the algorithm, perhaps calling it
;; from many other places in DO-STUFF besides the one I've shown.
? (defun do-stuff (...)
    ...
    (process-blah-using-algorithm-zark ...)
    ...)
DO-STUFF
;; Try it out...
? (do-stuff ...)
[results based upon process-blah-using-algorithm-zark]
;; OK, this looks good.  But I think I'll clean up the
;; appearance of DO-STUFF by defining an abbreviation
;; for that really long core algorithm name.
? (defsynonym process-blah-using-algorithm-zark proc)
PROC
;; Now I'll rewrite DO-STUFF to use the abbreviation.
? (defun do-stuff (...)
    ...
    (proc ...)
    ...)
DO-STUFF
;; And make sure it still works.
? (do-stuff ...)
[results based upon process-blah-using-algorithm-zark]

... Some time later ...

;; Oh, here's a better core algorithm.
? (defun process-blah-using-algorithm-glonkfarkle (...) ...)
PROCESS-BLAH-USING-ALGORITHM-GLONKFARKLE
;; I'll change the synonym for PROC to 'be' the new algorithm.
? (defsynonym process-blah-using-algorithm-glonkfarkle proc)
PROC

... Some time later ...

;; Time to use DO-STUFF again...
? (do-stuff ...)
[results based upon process-blah-using-algorithm-zark]
;; Hey!! These results don't seem to use the new algorithm.
;; What could be wrong?  The code LOOKS right...
```

The problem, of course, is that the second use of `DEFSYNONYM` redefined the `PROC` macro, and I didn't notice that `DO-STUFF` needed to be recompiled to pick up the changed definition.

> My advice: Don't try to be clever by using macros like `DEFSYNONYM`. Stick with descriptive names that are as long as necessary, and use an editor that supports symbol completion (see Chapter 27). Remember, there's only one way to *not* abbreviate a name; using abbreviations increases the chance that you'll use the wrong one.

Tricks of the trade: elude capture using GENSYM

You have to be be careful when you define a macro that introduces new variables in its expansion. The `REPEAT` macro, below, offers us a shorthand way of repeating a body of code a certain number of times.

```
? (defmacro repeat (times &body body)
   `(dotimes (x ,times)
      ,@body))
REPEAT
? (repeat 3 (print 'hi))

HI
HI
HI
NIL
```

This seems to do the right thing, but the variable `X` is going to cause problems. The following example *should* give us the same results as the last example.

```
? (setq x 'hi)
HI
? x
HI
? (repeat 3 (print x))

0
1
2
NIL
```

The variable `X` in the macro expander `shadowed` the global `X` that we tried to reference in the body. Another way to say this is that `X` is *free* in the body of the `REPEAT` form, but it was *captured* by the definition of `X` in the macro expander; this prevents the body form from reaching the intended variable `X`.

The obvious solution is to use a different variable name in the macro expander -- one that won't conflict with any name we'll ever use in our code that calls the `REPEAT` macro. You might think that some kind of naming convention would work, but there's always the chance that some programmer will come along later and violate the convention. We need a foolproof approach.

Lisp provides a `GENSYM` function to generate symbols that are *guaranteed* to be unique. No programmer can ever write a symbol name that conflicts with a symbol created by `GENSYM`. Here is how we use `GENSYM` to create a name for the variable needed in the macro expander for the `REPEAT` macro.

```
? (defmacro repeat (times &body body)
    (let ((x (gensym)))
      `(dotimes (,x ,times)
         ,@body)))
REPEAT
? x
HI
? (repeat 3 (print x))

HI
HI
HI
NIL
```

With this new `REPEAT` macro, we compute a new symbol in the `LET` form, and substitute this symbol in the macro expander form. To see why this works, let's look at an expansion:

```
? (macroexpand-1 '(repeat 5 (print x))
(DOTIMES (#:G8524 5) (PRINT X))
```

`#:G8524` is a unique *uninterned* symbol. You can see that it's uninterned by the `#:` prefix. But how does Lisp guarantee the uniqueness of this symbol? The Lisp reader guarantees that any symbol it reads with the `#:` prefix is unique. Compare the following:

```
? (eq 'a 'a)
T
? (eq '#:a '#:a)
NIL
```

Even though the `#:A` symbols print the same, they are different.

Generating variable names to be used in macro expanders has another application. This next macro definition has a subtle problem:

```
? (defmacro cube (n)
    `(* ,n ,n ,n))
CUBE
? (cube 3)
27
? (let ((n 2))
    (cube (incf n)))
60
```

In the second case, `(INCF N)` should have provided the value 3 to `CUBE` and the result should have been identical to the first test. Let's take a look at the expansion again, to see what happened.

```
? (macroexpand-1 '(cube (incf n)))
(* (INCF N) (INCF N) (INCF N))
```

The problem is obvious: `CUBE`'s argument, `(INCF N)` is being evaluated multiple times. As a rule, this is a bad thing to do, because it violates our assumptions about the way Lisp evaluates forms. We fix this problem by arranging for the macro expander to evaluate `CUBE`'s argument just once.

```
? (defmacro cube (n)
    (let ((x (gensym)))
      `(let ((,x ,n))
         (* ,x ,x ,x))))
CUBE
? (let ((n 2))
    (cube (incf n)))
27
```

We created a unique symbol outside of the macro expander, then used this symbol in the expander as the name of a variable to hold the result of evaluating `CUBE`'s argument. The `LET` form in the macro expander is the *only* place where `CUBE`'s argument is referenced, so it gets evaluated exactly once for each call to `CUBE`.

Macros vs. inlining

Lisp allows functions to be inlined by the compiler. In other words, rather than compiling a call to the function, the compiler may substitute the function's body, thus saving the overhead of a function call. Substituting the function's body is generally expensive in terms of space, since a function body's code is usually longer than the code of its calling sequence.

It's important to understand that Lisp *allows* functions to be inlined. Like all other declarations -- save the `SPECIAL` declaration -- an `INLINE` declaration may be treated as advisory or ignored entirely by the compiler.

Here are some examples of how to inline a function. In the first case, function `F` is inlined everywhere it is used (assuming that the compiler supports inlining). In the second case, function `P` is compiled with information to support inlining, but is only inlined in the presence of a declaration, as in function `Q`.

```
; Case 1 -- F may always be inlined
(declaim (inline f))
(defun f (...) ...)
```

```
(defun g (...)
  (f ...)
  ...)

(defun h (...)
  (f ...)
  ...)

; Case 2 - P may be inlined only following a declaration
(declaim (inline p))
(defun p (...) ...)
(declaim (notinline p))

(defun q (...)
  (declare (inline p))
  (p ...) ; inlined
  ...)

(defun r (...)
  (p ...) ; not inlined
  ...)
```

Macros may be used in place of `INLINE` declarations for cases where code absolutely *must* be inlined despite the presence (or absence) of compiler support for inlining.

In general, though, you should use macros for language extension, and not for efficiency hacks. The risk of forgetting to recompile after having changed a macro definition can cause hard-to-find bugs that will waste a lot of your development effort.

> My advice: Don't use macros as a substitute for inlining unless you can find no other way to achieve desired performance; of course, such efforts should be guided by the results of profiling your code (see Chapter 28) and preferably only when your code is already stable and debugged. You should also reexamine your decision with each new release of your Lisp compiler, and whenever you port your program to another Lisp platform.

Contents | Cover
Chapter 19 | Chapter 20 | Chapter 21

Copyright © 1995-2001, David B. Lamkins
All Rights Reserved Worldwide

This book may not be reproduced without the written consent of its author. Online distribution is restricted to the author's site.

Chapter 21 - Fancy Tricks with Function and Macro Arguments

We've already seen (in Chapter 4) how `&OPTIONAL` parameters can reduce the number of arguments that you have to supply for the most common calls of a function. In this chapter we'll look at additional language features that let you declare named (keyword) parameters and provide default values for unspecified parameters. We'll also take a look at structured argument lists, which let you group related parameters for clarity.

Keywords let you name your parameters

Sometimes you'll want to define a function (or macro) that works just fine with a small list of arguments, but can be extended in useful -- and obvious, I hope -- ways through the addition of extra arguments. But you'd rather not specify *all* of the arguments *all* of the time. We've already seen keyword arguments used in Chapter 13 with the sequence functions, and in Chapter 19 with the stream functions.

You can use keyword arguments for your own functions or macros by adding a `&key` marker to the lambda list. The general form (also used for `DEFMACRO`) is:

```
(defun name (req-arg ... &key key-arg)
  ...)
```

All of the required arguments (*req-arg*) must precede the `&KEY` marker. The *key-arg*s name the variable that you'll reference from within your function's definition; the same *key-arg* name in the keyword package (i.e. preceded with a colon) is used in the call as a prefix for the keyword value.

```
? (defun keyword-sample-1 (a b c &key d e f)
    (list a b c d e f))
KEYWORD-SAMPLE-1
? (keyword-sample-1 1 2 3)
(1 2 3 NIL NIL NIL)
? (keyword-sample-1 1 2 3 :d 4)
(1 2 3 4 NIL NIL)
? (keyword-sample-1 1 2 3 :e 5)
(1 2 3 NIL 5 NIL)
? (keyword-sample-1 1 2 3 :f 6 :d 4 :e 5)
(1 2 3 4 5 6)
```

You'll notice from the last sample that keyword parameters may be listed in any order. However, as in their lambda list declaration, all keyword parameters must follow all required parameters.

Default values for when you'd rather not say

Any keyword parameter that you don't specify in a call receives a `NIL` default value. You can change the default using a variation of the keyword argument declaration: instead of just the argument name, specify `(`*name default*`)`, like this:

```
? (defun keyword-sample-2 (a &key (b 77) (c 88))
    (list a b c))
KEYWORD-SAMPLE-2
? (keyword-sample-2 1)
(1 77 88)
```

```
? (keyword-sample-2 1 :c 3)
(1 77 3)
```

You can also find out whether a keyword parameter was specified in the call, even if it was specified using the default value. The keyword argument declaration looks like this: `(`*name default arg-supplied-p*`)`, where *arg-supplied-p* is the name of a variable that your function's definition reads as `NIL` only if no argument is supplied in the call.

```
? (defun keyword-sample-3 (a &key (b nil b-p) (c 53 c-p))
    (list a b b-p c c-p))
KEYWORD-SAMPLE-2
? (keyword-sample-3 1)
(1 NIL NIL 53 NIL)
? (keyword-sample-3 1 :b 74)
(1 74 T 53 NIL)
? (keyword-sample-3 1 :b nil)
(1 NIL T 53 NIL)
? (keyword-sample-3 1 :c 9)
(1 NIL NIL 9 T)
```

Default values and supplied-p variable can also be used with `&OPTIONAL` parameters.

```
? (defun optional-sample-1 (a &optional (b nil b-p))
    (list a b b-p))
OPTIONAL-SAMPLE-1
? (optional-sample-1 1)
(1 NIL NIL)
? (optional-sample-1 1 nil)
(1 NIL T)
? (optional-sample-1 1 2)
(1 2 T)
```

If you use both `&OPTIONAL` and `&KEY` parameters, all of the optional parameters must precede all of the keyword parameters, both in the declaration and the call. Of course, the required parameters must always appear before all other parameters.

```
? (defun optional-keyword-sample-1 (a &optional b c &key d e)
    (list a b c d e))
OPTIONAL-KEYWORD-SAMPLE-1
? (optional-keyword-sample-1 1)
(1 NIL NIL NIL NIL)
? (optional-keyword-sample-1 1 2)
(1 2 NIL NIL NIL)
? (optional-keyword-sample-1 1 2 3)
(1 2 3 NIL NIL)
? (optional-keyword-sample-1 1 2 3 :e 5)
(1 2 3 NIL 5)
```

When you define both `&OPTIONAL` and `&KEY` arguments, the call must include values for *all* of the optional parameters if it specifies *any* keyword parameters, as in the last sample, above. Look at what can happen if you omit some optional parameters:

```
? (defun optional-keyword-sample-2 (a &optional b c d &key e f)
    (list a b c d e f))
OPTIONAL-KEYWORD-SAMPLE-2
? (optional-keyword-sample-2 1 2 :e 3)
(1 2 :E 3 NIL NIL)
```

Even though a Common Lisp function (`READ-FROM-STRING`) uses both optional and keyword arguments, you should *not* do the same when you define your own functions or macros.

Add some structure to your macros by taking apart arguments

You can use *destructuring* to create groups of parameters for macros.

```
? (defmacro destructuring-sample-1 ((a b) (c d))
    `(list ',a ',b ',c ',d))
DESTRUCTURING-SAMPLE-1
? (destructuring-sample-1 (1 2) (3 4))
(1 2 3 4)
```

You can use all the usual techniques within each group.

```
? (defmacro destructuring-sample-2 ((a &key b) (c &optional d))
    `(list ',a ',b ',c ',d))
DESTRUCTURING-SAMPLE-2
? (destructuring-sample-2 (1) (3))
(1 NIL 3 NIL)
? (destructuring-sample-2 (1 :b 2) (3))
(1 2 3 NIL)
? (destructuring-sample-2 (1) (3 4))
(1 NIL 3 4)
```

And the groupings can even be nested.

```
? (defmacro destructuring-sample-3 ((a &key b) (c (d e) &optional f))
    `(list ',a ',b ',c ',d ',e ',f))
DESTRUCTURING-SAMPLE-3
? (destructuring-sample-3 (1) (3 (4 5)))
(1 NIL 3 4 5 NIL)
```

Destructuring is commonly used to set off a group of arguments or declarations from the body forms in a macro. Here's an extended example, `WITH-PROCESSES`, that expects a name, a list of a variable name (pid) and a process count (num-processes), and a list of another variable name (work-item) and a list of elements to process (work-queue). All of these arguments are grouped before the body forms.

```
? (defmacro with-processes ((name
                             (pid num-processes)
                             (work-item work-queue)) &body body)
    (let ((process-fn (gensym))
          (items (gensym))
          (items-lock (gensym)))
      `(let ((,items (copy-list ,work-queue))
             (,items-lock (make-lock)))
         (flet ((,process-fn (,pid)
                  (let ((,work-item nil))
                    (loop
                      (with-lock-grabbed (,items-lock)
                        (setq ,work-item (pop ,items)))
                      (when (null ,work-item)
                        (return))
                      ;;(format t "~&running id ~D~%" ,pid)
                      ,@body))))
           (dotimes (i ,num-processes)
             ;;(format t "~&creating id ~D~%" ,id)
             (process-run-function
              (format nil "~A--~D" ,name i)
              #',process-fn
              i))))))
WITH-PROCESSES
```

Processes are not part of the ANSI Common Lisp standard, but are present in almost every implementation. (We'll revisit processes in Chapter 32, along with some other common language extensions.) The code shown above works with Macintosh Common Lisp, whose process interface is very similar to that found on the Symbolics Lisp Machines of days past.

I'll describe a few key portions of the macro expander in case you want to figure out what's going on; if you'd rather just see how the macro gets called, you can skip the rest of this paragraph. The `FLET` form defines a function. In this case, the function defined by `FLET` will be used to do the actual work within a Lisp process -- grab a lock on the work queue, remove an item, release the lock, then process the item using the body forms. The `PROCESS-RUN-FUNCTION` creates a Lisp process with a given name (generated by the `FORMAT` form) and a function to execute. The `WITH-PROCESSES` macro creates `NUM-PROCESSES` Lisp processes (named *name-#*) and within each process executes the `BODY` forms with `PID` bound to the process number and `WORK-ITEM` bound to some element of `WORK-QUEUE`. The processes terminate themselves once the work queue has been consumed.

Here's an example of how we call `WITH-PROCESSES`. The parameters are `"Test"` (used for the process names), `(id 3)` (the variable bound to the process ID within a process, and the number of processes to create), and `(item '(1 2 ... 15 16)` (the variable bound to an individual work item within a process, and the list of items to be consumed by the processes). The `FORMAT` and `SLEEP` forms comprise the body of the processes, and the final argument to the `WITH-PROCESSES` macro call.

```
? (with-processes ("Test"
                   (id 3)
                   (item '(1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16)))
    (format t "~&id ~D item ~A~%" id item)
    (sleep (random 1.0)))
NIL
id 0 item 1
id 1 item 2
id 2 item 3
id 1 item 4
id 1 item 5
id 0 item 6
id 2 item 7
id 0 item 8
id 2 item 9
id 1 item 10
id 2 item 11
id 0 item 12
id 0 item 13
id 1 item 14
id 2 item 15
id 0 item 16
```

The form returns NIL almost immediately, but the created processes run for a while to produce the output that follows. The "item" numbers follow an orderly progression as they are consumed from the work queue, but the "id" numbers vary according to which process actually consumed a particular item.

Destructuring is a useful tool for macros, but you can't use it in the lambda list of a function. However, you can destructure a list from within a function via `DESTRUCTURING-BIND`.

```
? (destructuring-bind ((a &key b) (c (d e) &optional f))
                      '((1 :b 2) (3 (4 5) 6))
    (list a b c d e f))
(1 2 3 4 5 6)
```

Contents | Cover
Chapter 20 | Chapter 21 | Chapter 22

Copyright © 1995-2001, David B. Lamkins
All Rights Reserved Worldwide

This book may not be reproduced without the written consent of its author. Online distribution is restricted to the author's site.

Chapter 22 - How to Find Your Way Around, Part 3

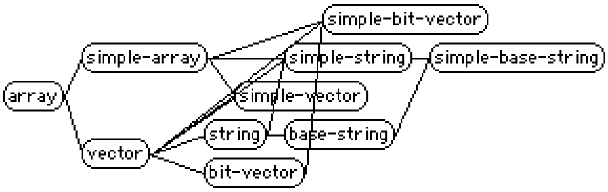
In Chapter 10 we learned about two functions you can use to examine Lisp objects: `DESCRIBE` and `INSPECT`. These are available in every implementation, so you should learn how to use them just in case you find yourself sitting at a console in front of a new and unknown Lisp system someday.

Some Lisp systems offer additional tools that aren't part of ANSI Common Lisp. Sometimes the extra tools are built in, and other times they're provided by the vendor but not installed by default. In this chapter I'll tell you a bit about these tools, so you'll know what to look for.

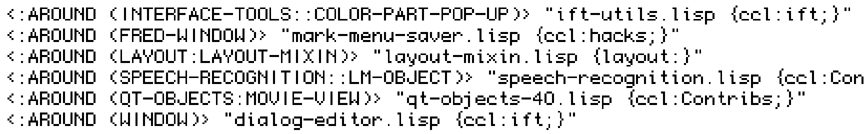
In addition to exploring the structure and relationships of objects, it's sometimes useful to "hook in" to the behavior of certain functions. You might just want to know when -- or whether -- a function is called with certain arguments. `TRACE` is always at your disposal (see Chapter 16), but you may only be interested in one particular call out of thousands; generating the trace output (never mind sifting through it later) can be very time consuming. For cases like this, some Lisp implementations let you *advise* an existing function without changing its source code.

Class and method browsers help you find your way in a sea of objects

When you program a large system using CLOS, especially if the system evolves over time as so many do, you'll need a tool to help you examine the relationships between classes. Some Lisp systems provide a browser that displays these relationships graphically. Here's an example of what my system's browser displays as the subclasses of `ARRAY`:



Another useful tool, the method browser, lets you explore all of the specializations of a particular method. The browser may let you filter on particular qualifiers (we learned about qualifiers in Chapter 14), and will almost certainly give you navigational access to selected method definitions. Here's a list from the method browser in my Lisp system; I asked for a list of all *INITIALIZE-INSTANCE* methods having the `:AROUND` qualifier:



```
<:AROUND (INTERFACE-TOOLS::COLOR-PART-POP-UP)> "ift-utils.lisp {ccl:ift;}"
<:AROUND (FRED-WINDOW)> "mark-menu-saver.lisp {ccl:hacks;}"
<:AROUND (LAYOUT:LAYOUT-MIXIN)> "layout-mixin.lisp {layout:}"
<:AROUND (SPEECH-RECOGNITION::LM-OBJECT)> "speech-recognition.lisp {ccl:Con
<:AROUND (QT-OBJECTS:MOVIE-VIEW)> "qt-objects-40.lisp {ccl:Contribs;}"
<:AROUND (WINDOW)> "dialog-editor.lisp {ccl:ift;}"
```

Remember that methods do not belong to classes; that's why we have separate browsers. (Some browsers give you the ability to navigate the coincidental connections between classes and methods, such as by examining the classes used to specialize arguments to a method.)

ADVISE lets you modify a function's behavior without changing the function

Some Lisp systems include an `ADVISE` capability. This lets you intercept calls to existing functions. You can provide code that examines (and perhaps alters) the function's arguments and results. `ADVISE` has many uses, most of them invented on the spur of the moment. However, one common use of `ADVISE` is to implement a `TRACE` or `BREAK` that is conditioned upon particular arguments or results.

The syntax and options for `ADVISE` vary from system to system. Here's an example of defining advice in one particular implementation:

```
(advise fibonacci
        (when (zerop (first arglist)) (break))
        :when :before
        :name :break-on-zero)
```

This example shows how to advise a `FIBONACCI` function by adding code that breaks into the debugger when `FIBONACCI`'s first argument is zero. Note that we do not need any knowledge of or access to the source code of `FIBONACCI` in order to add this advice.

This particular implementation of `ADVISE` binds a list of all of the advised function's arguments into a variable named `ARGLIST`. The keyword arguments declare that the advice form, `(WHEN (ZEROP (FIRST ARGLIST)) (BREAK))`, is to be executed *before* each call to `FIBONACCI`. The advice has the name `:BREAK-ON-ZERO`; this name is used when removing advice (typically via an `UNADVISE` form) or when redefining the behavior of a particular advice.

WATCH lets you open a window on interesting variables

A *watch* tool, found less commonly on Lisp systems, allows you to display the current state of a variable as your program runs. The details vary widely. Implementations may give you sampled real-time display, or may slow the program down in order to give you an accurate display of each change. Sometimes the watcher is integrated with a debugger or stepper, and other times it is an independent tool. Consult your vendor's documentation to learn whether your Lisp system has a watch tool.

Contents | Cover
Chapter 21 | Chapter 22 | Chapter 23

Copyright © 1995-2001, David B. Lamkins
All Rights Reserved Worldwide

This book may not be reproduced without the written consent of its author. Online distribution is restricted to the author's site.

Chapter 23 - To Err is Expected; To Recover, Divine

In this chapter you'll learn how to create your own error detection, reporting and recovery mechanisms. A good error handling strategy can give your program the ability to gracefully handle both expected and unexpected errors without failing or losing critical data.

Signal your own errors and impress your users

One of the most common failings of computer programs is the failure to report failures in a meaningful way. If some input is out of the expected range, or if a calculation exceeds the capabilities of the program, or if communication does not succeed with some external device, a poorly-written program will simply "roll over and die" with a cryptic error message related to hidden details of the program's implementation. In theory, it's nice to be able to construct programs without limits; the dynamic nature of Lisp certainly enables this practice.

But in almost every non-trivial program there will always arise some fatal situation that can be anticipated by the programmer but not addressed by the program. It is precisely for these situations that Lisp provides the `ERROR` function. `ERROR` expects a format string and arguments. (We've seen `FORMAT` briefly in Chapter 4, and will examine it in detail in Chapter 24.) `ERROR` gives your program a standard way to announce a fatal error. You simply compose an appropriate message using the format string and (optional) arguments, and `ERROR` takes care of the rest.

```
? (defun divide (numerator denominator)
    (when (zerop denominator)
      (error "Sorry, you can't divide by zero."))
    (/ numerator denominator))
DIVIDE
? (divide 4 3)
4/3
? (divide 1 0)
Error: Sorry, you can't divide by zero.
```

Your program never returns from the call to `ERROR`. Instead, the Lisp debugger will be entered. You'll have an opportunity to examine the cause of the error while in the debugger, but you will not be able to resume your program's execution. This makes `ERROR` a rather extreme response to a problem detected by your program. Later, we'll see how to report problems *and* give the user an opportunity to correct the problem. We'll even see how errors can be handled automatically.

Categorize errors using Conditions

> Note: If you have a really old Lisp system, it may not include an implementation of conditions. If so, this section and the following one may not be of much use to you, except to point out what your Lisp system lacks as compared to the current standard.

An error is just a condition that requires some kind of correction before your program may continue. The error may be corrected by having the program's user interact with the debugger, or through the intervention of a handler (as we'll see later in this chapter).

A condition is just some exceptional event that happens in your program. The event may be due to an error, or it could be something of interest that happens while your program runs. For example, a program that writes entries to a log file on disk might call a routine that handles the record formatting and writing. The logging routine might periodically check the amount of space available on disk and signal a condition when the disk becomes ninety percent full. This is not an error, because the logger won't fail under this condition. If your program ignores the "almost-full" message from the logger, nothing bad will happen. However, your program may wish to do something useful with the information about the available disk space, such as archiving the log to a different device or informing the user that some corrective action may be needed soon.

Now we know the distinction between conditions and errors. For now, we're going to focus our attention on the tools Lisp provides for handling errors. Later, we'll look at how your program can signal and handle conditions.

You could report errors using format strings as described above. But for the sake of consistency and maintainability, you'll probably want to create different classifications of errors. That way, you can change the presentation of an entire class of errors without searching your program code to change all of the similar format strings.

A `condition` represents some *exceptional situation* which occurs during the execution of your program. An error is a kind of condition, but not all conditions are errors. The next section will cover this distinction in greater detail.

You can use `DEFINE-CONDITION` to create type hierarchies for conditions in much the same way that you use `DEFCLASS` to create type hierarchies for your program's data.

```
? (define-condition whats-wrong (error)
                    ((what :initarg :what :initform "something" :reader what))
    (:report (lambda (condition stream)
               (format stream "Foo! ~@(~A~) is wrong."
                       (what condition))))
    (:documentation "Tell the user that something is wrong."))
WHATS-WRONG
? (define-condition whats-wrong-and-why (whats-wrong)
                    ((why :initarg :why :initform "no clue" :reader why))
    (:report (lambda (condition stream)
               (format stream "Uh oh! ~@(~A~) is wrong. Why? ~@(~A~)."
                       (what condition)
                       (why condition)))))
WHATS-WRONG-AND-WHY
? (error 'whats-wrong-and-why)
Error: Uh oh! Something is wrong. Why? No clue.
? (error 'whats-wrong-and-why
         :what "the phase variance"
         :why "insufficient tachyon flux")
Error: Uh oh! The phase variance is wrong. Why? Insufficient tachyon flux.
? (define-condition whats-wrong-is-unfathomable (whats-wrong-and-why)
                    ()
    (:report (lambda (condition stream)
               (format stream "Gack! ~@(~A~) is wrong for some inexplicable reason."
                       (what condition)))))
WHATS-WRONG-IS-UNFATHOMABLE
? (error 'whats-wrong-is-unfathomable)
Error: Gack! Something is wrong for some inexplicable reason.
```

As you can see, conditions have parents, slots and options just like classes. The `:REPORT` option is used to generate the textual presentation of a condition. The `:DOCUMENTATION` option is for the benefit of the programmer; you can retrieve a condition's documentation using `(DOCUMENTATION '`*`condition-name`*` 'type)`.

> ANSI Common Lisp also allows a `:DEFAULT-INITARGS` option. Some Lisp systems still base their implementation of conditions on the description found in Guy Steele's "Common Lisp: The Language, 2nd Edition" (CLtL2); these implementations do not have a `:DEFAULT-INITARGS` option.

If you've compared the `ERROR` calls in this section to those of the previous section, you're probably wondering how both a string and a symbol can designate a condition. If you pass a symbol to `ERROR` , it constructs a condition using `MAKE-CONDITION` (analogous to `MAKE-INSTANCE` for CLOS objects); the symbol designates the type of the condition, and the arguments are used to initialize the condition. If you pass a format string to `ERROR`, the format string and its arguments become initialization options for the construction of a condition of type `SIMPLE-ERROR`.

Of course, you can also pass an instantiated condition object to `ERROR`:

```
? (let ((my-condition (make-condition 'simple-error
                                    :format-control "Can't do ~A."
                                    :format-arguments '(undefined-operation))))
  (error my-condition))
Error: Can't do UNDEFINED-OPERATION.
```

> Lisp systems designed according to CLtL2 will use `:FORMAT-STRING` in place of `:FORMAT-CONTROL`.

Recover from Conditions using Restarts

In this final section, we'll see how to recover from errors. The simplest forms involve the use of `CERROR` or `ASSERT`.

```
? (progn (cerror "Go ahead, make my day."
                 "Do you feel lucky?")
         "Just kidding")
Error: Do you feel lucky?
Restart options:
  1: Go ahead, make my day.
  2. Top level
```

> The "Restart options" list shown in this and the following examples is typical, but not standard. Different Lisp systems will present restart information in their own ways, and may add other built in options.

`CERROR` has two required arguments. The first argument is a format control string that you'll use to tell the program's user what will happen upon `continuing` from the error. The second argument is a condition designator (a format control string, a symbol that names a condition, or a condition object -- see above) used to tell the program's user about the error.

The rest of `CERROR`'s arguments, when present, are used by the the format control strings *and* -- when the second argument is a symbol that names a condition type -- as keyword arguments to `MAKE-CONDITION` for that type. In either case, you have to construct the format control strings so that they address the proper arguments. The `FORMAT` directive ~*n** can be used to skip *n* arguments (*n* is 1 if omitted).

```
? (defun expect-type (object type default-value)
    (if (typep object type)
      object
      (progn
        (cerror "Substitute the default value ~2*~S."
                "~S is not of the expected type ~S."
                object type default-value)
        default-value)))
EXPECT-TYPE
? (expect-type "Nifty" 'string "Bear")
"Nifty"
? (expect-type 7 'string "Bear")
Error: 7 is not of the expected type STRING.
Restart options:
  1: Substitute the default value "Bear".
  2. Top level
  ? 1
"Bear"
```

Notice how the first format control string uses only the third format argument: `DEFAULT-VALUE`. It skips the first two format arguments with the `~2*` directive. You do similar things if the arguments are keyword initializer arguments when you provide a symbol as the second argument to `CERROR`; the only difference is that you have to count the keywords `and` the values when deciding how many arguments to skip. Here's the previous example, written with a designator for a condition of type `EXPECT-TYPE-ERROR` instead of a format control string. Note how we skip five arguments to get to the `DEFAULT-VALUE`. Note also the use of `:ALLOW-OTHER-KEYS T`, which permits us to add the `:IGNORE DEFAULT-VALUE` keyword argument which is not expected as an initialization argument for the `EXPECT-TYPE-ERROR` condition; without this, we'd get an error for the unexpected keyword argument.

```
? (define-condition expect-type-error (error)
                    ((object :initarg :object :reader object)
                     (type :initarg :type :reader type))
    (:report (lambda (condition stream)
               (format stream "~S is not of the expected type ~S."
                       (object condition)
                       (type condition)))))
EXPECT-TYPE-ERROR
? (defun expect-type (object type default-value)
    (if (typep object type)
      object
      (progn
        (cerror "Substitute the default value ~5*~S."
                'expect-type-error
                :object object
                :type type
                :ignore default-value
                :allow-other-keys t)
        default-value)))
EXPECT-TYPE
? (expect-type "Nifty" 'string "Bear")
"Nifty"
? (expect-type 7 'string "Bear")
Error: 7 is not of the expected type STRING.
Restart options:
  1: Substitute the default value "Bear".
  2. Top level
  ? 1
"Bear"
```

`ASSERT` is ideal for those situations where your program's state must pass some test -- an *assertion*. In its simplest form, `ASSERT` does only that.

```
? (defun my-divide (numerator denominator)
    (assert (not (zerop denominator)))
    (/ numerator denominator))
MY-DIVIDE
? (my-divide 3 0)
Error: Failed assertion (NOT (ZEROP DENOMINATOR))
Restart options:
  1. Retry the assertion
  2. Top level
```

This report is correct, but nor particularly useful; your program would have signalled a `DIVISION-BY-ZERO` error without the `ASSERT`. What *would* be helpful is the ability to correct the offending value -- the zero denominator, in this case -- and continue from the error. `ASSERT`'s optional second argument lets you list places whose values you might want to change to correct the problem.

```
? (defun my-divide (numerator denominator)
    (assert (not (zerop denominator)) (numerator denominator))
    (/ numerator denominator))
MY-DIVIDE
? (my-divide 3 0)
Error: Failed assertion (NOT (ZEROP DENOMINATOR))
Restart options:
  1. Change the values of some places, then retry the assertion
  2. Top level
  ? 1
  Value for NUMERATOR: 3
  Value for DENOMINATOR 0.5
6.0
```

Of course, the choice of values to set is up to you. I used both `NUMERATOR` and `DENOMINATOR` in the example to emphasize the fact that the list of places does not have to be just the variables tested in the assertion. (However, at least `one` of the places must affect the result of the assertion.)

One last refinement to `ASSERT` lets you specify your own message to use when an assertion fails. By default, `ASSERT` may display the test form, but it is not required to do so. By specifying a condition designator and arguments following the list of places, you can be assured that you know what message will be printed upon an assertion failure.

```
? (defun my-divide (numerator denominator)
    (assert (not (zerop denominator)) (numerator denominator)
            "You can't divide ~D by ~D." numerator denominator)
    (/ numerator denominator))
MY-DIVIDE
? (my-divide 3 0)
Error: You can't divide 3 by 0.
Restart options:
  1. Change the values of some places, then retry the assertion
  2. Top level
  ? 1
  Value for NUMERATOR: 3
  Value for DENOMINATOR 2
3/2
```

You can use `HANDLER-BIND` and `SIGNAL` to process exceptions in your program. Here's an extended example based upon this chapter's earlier description of how a program might use conditions to report on disk space availability.

```
? (define-condition high-disk-utilization ()
                    ((disk-name :initarg :disk-name :reader disk-name)
                     (current :initarg :current :reader current-utilization)
                     (threshold :initarg :threshold :reader threshold))
    (:report (lambda (condition stream)
               (format stream "Disk ~A is ~D% full; threshold is ~D%."
                       (disk-name condition)
                       (current-utilization condition)
                       (threshold condition)))))
HIGH-DISK-UTILIZATION
? (defun get-disk-utilization (disk-name)
    ;; for this example, we'll just return a fixed value
    93)
GET-DISK-UTILIZATION
? (defun check-disk-utilization (name threshold)
    (let ((utilization (disk-utilization name)))
      (when (>= utilization threshold)
        (signal 'high-disk-utilization
                :disk-name name
                :current utilization
                :threshold threshold))))
CHECK-DISK-UTILIZATION
? (defun log-to-disk (record name)
    (handler-bind ((high-disk-utilization
                    #'(lambda (c)
                        (when (y-or-n-p "~&~A Panic?" c)
                          (return-from log-to-disk nil)))))
      (check-disk-utilization name 90)
      (print record))
    t)
LOG-TO-DISK
? (log-to-disk "Hello" 'disk1)
Disk DISK1 is 93% full; threshold is 90%. Panic? (y or n)  n
"Hello"
T
? (log-to-disk "Goodbye" 'disk1)
Disk DISK1 is 93% full; threshold is 90%. Panic? (y or n)  y
NIL
? (check-disk-utilization 'disk1 90)
NIL
```

Notice that the condition signalled by `CHECK-DISK-UTILIZATION` has an effect only when a handler is established for the `HIGH-DISK-UTILIZATION` condition. Because of this, you can write exception signalling code without foreknowledge that the client will provide a handler. This is most useful when the exception provides information about the running program, but is not an error if left unhandled.

In the next example, we'll extend the restart options available to `CERROR`. `RESTART-BIND` defines, for each new restart, the message to be printed by the restart user interface and a function to be executed when the user chooses the restart.

```
? (define-condition device-unresponsive ()
                    ((device :initarg :device :reader device))
    (:report (lambda (condition stream)
               (format stream "Device ~A is unresponsive."
                       (device condition)))))
DEVICE-UNRESPONSIVE
? (defun send-query (device query)
    (format t "~&Sending ~S ~S~%" device query))
SEND-QUERY
? (defun accept-response (device)
    ;; For the example, the device always fails.
    nil)
ACCEPT-RESPONSE
? (defun reset-device (device)
    (format t "~&Resetting ~S~%" device))
RESET-DEVICE
? (defun query-device (device)
    (restart-bind ((nil #'(lambda () (reset-device device))
                        :report-function
                        #'(lambda (stream)
                            (format stream "Reset device.")))
                   (nil #'(lambda ()
                            (format t "~&New device: ")
                            (finish-output)
                            (setq device (read)))
                        :report-function
                        #'(lambda (stream)
                            (format stream "Try a different device.")))
                   (nil #'(lambda ()
                            (return-from query-device :gave-up))
                        :report-function
                        #'(lambda (stream)
                            (format stream "Give up."))))
      (loop
        (send-query device 'query)
        (let ((answer (accept-response device)))
          (if answer
            (return answer)
            (cerror "Try again."
                    'device-unresponsive :device device))))))
QUERY-DEVICE
? (query-device 'foo)
Sending FOO QUERY
Error: Device FOO is unresponsive.
Restart options:
  1. Try again.
  2. Reset device.
  3. Try a different device.
  4. Give up.
  5. Top level
  ? 1
Sending FOO QUERY
Error: Device FOO is unresponsive.
Restart options:
  1. Try again.
  2. Reset device.
  3. Try a different device.
  4. Give up.
  5. Top level
  ? 2
Resetting FOO
Restart options:
  1. Try again.
  2. Reset device.
  3. Try a different device.
  4. Give up.
  5. Top level
  ? 1
Sending FOO QUERY
Error: Device FOO is unresponsive.
Restart options:
  1. Try again.
  2. Reset device.
  3. Try a different device.
  4. Give up.
  5. Top level
  ? 3
New device: bar
Restart options:
  1. Try again.
  2. Reset device.
  3. Try a different device.
  4. Give up.
  5. Top level
  ? 1
Error: Device BAR is unresponsive.
Restart options:
  1. Try again.
  2. Reset device.
  3. Try a different device.
  4. Give up.
  5. Top level
  ? 4
:GAVE-UP
```

The "Try again" restart is established by the `CERROR` form; selecting this restart lets the program continue from the `CERROR` form. The "Reset device", "Try a different device", and "Give up" restarts are created within the `RESTART-BIND` form; choosing one of these executes the associated function. Of the restarts defined within the `RESTART-BIND`, only the "Give up" restart transfers control out of the `CERROR` form -- the others return control to `CERROR` to again display the menu of restart choices.

Now you've seen the basics of condition handlers and restarts. Lisp has additional built-in abstractions that extend these concepts. If you're interested, you should consult a Common Lisp reference.

There's one last thing you should know about handling conditions. As we saw earlier, `ERROR` causes your program to stop in the Lisp debugger. You can't continue past the call to `ERROR`, but most Lisp systems will let you back up, correct the problem that caused the error, and rerun that portion of the program. If you do the right thing, your program won't call `ERROR` again. This is an amazingly powerful tool to use during program development. But you don't want to expose your users to that kind of experience -- they won't be as impressed by the Lisp debugger as you are.

To protect your users from the debugger, you can wrap portions of your program in an `IGNORE-ERRORS` form.

```
? (ignore-errors
   (error "Something bad has happened.")
   (print "Didn't get here."))
NIL
#<SIMPLE-ERROR #x42B26B6>
? (ignore-errors
   (* 7 9))
63
```

If an error occurs within an `IGNORE-ERRORS` form, program execution ends at that point, and `IGNORE-ERRORS` returns two values: `NIL` and the condition signalled by `ERROR`.

You should use `IGNORE-ERRORS` judiciously. Use it only to wrap forms for which you can't otherwise provide handlers. Note, too, that the values returned by `IGNORE-ERRORS` are not very informative. But you can decode the second return value to print the actual error message.

```
? (defmacro report-error (&body body)
    (let ((results (gensym))
          (condition (gensym)))
      `(let ((,results (multiple-value-list
                        (ignore-errors
                         ,@body))))
         (if (and (null (first ,results))
                  (typep (second ,results) 'condition)
                  (null (nthcdr 2 ,results)))
           (let ((,condition (second ,results)))
             (typecase ,condition
               (simple-condition
                (apply #'format t
                       (simple-condition-format-control ,condition)
                       (simple-condition-format-arguments ,condition)))
               (otherwise
                (format t "~A error." (type-of ,condition))))
             (values))
           (values-list ,results)))))
REPORT-ERROR
? (report-error (error "I feel like I'm losing my mind, Dave."))
I feel like I'm losing my mind, Dave.
? (report-error (+ 1 no-variable-by-this-name))
UNBOUND-VARIABLE error.
? (report-error (* 7 'f))
TYPE-ERROR error.
? (report-error (let ((n 1)) (/ 8 (decf n))))
DIVISION-BY-ZERO error.
? (report-error (* 2 pi)) ; not an error
6.283185307179586
? (report-error (values 1 2 3 4)) ; not an error
```

Contents | Cover
Chapter 22 | Chapter 23 | Chapter 24

Copyright © 1995-2001, David B. Lamkins
All Rights Reserved Worldwide

This book may not be reproduced without the written consent of its author. Online distribution is restricted to the author's site.

Chapter 24 - FORMAT Speaks a Different Language

Throughout this book, we've shown some simple uses of `FORMAT` without explanation. In this chapter, we'll inventory and describe `FORMAT`'s most useful capabilities, and a few of its more esoteric features.

FORMAT rhymes with FORTRAN, sort of...

`FORMAT` appears to have been inspired by FORTRAN's varied and capable function of the same name. But Lisp's `FORMAT` implements a programming language in its own right, designed expressly for the purposes of formatting textual output. `FORMAT` can print data of many types, using various decorations and embellishments. It can print numbers as words or -- for you movie buffs -- as Roman numerals. Columnar output is a breeze with `FORMAT`'s facilities for iterating over lists and advancing to specific positions on an output line. You can even make portions of the output appear differently depending upon the formatted variables. This chapter covers a representative portion of what `FORMAT` can do; you should consult a Lisp reference to get the full story.

Formatting

`FORMAT` expects a destination argument, a format control string, and a list of zero or more arguments to be used by the control string to produce formatted output.

Output goes to a location determined by the destination argument. If the destination is `T`, output goes to `*STANDARD-OUTPUT*`. The destination can also be a specific output stream.

There are two ways `FORMAT` can send output to a string. One is to specify `NIL` for the destination: `FORMAT` will return a string containing the formatted output. The other way is to specify a string for the destination; the string must have a fill pointer.

```
? (defparameter *s*
    (make-array 0
                :element-type 'character
                :adjustable t
                :fill-pointer 0))
""
? (format *s* "Hello~%")
NIL
? *s*
"Hello
"
? (format *s* "Goodbye")
NIL
? *s*
"Hello
Goodbye"
? (setf (fill-pointer *s*) 0)
0
? *s*
""
? (format *s* "A new beginning")
NIL
? *s*
"A new beginning"
```

The call to `MAKE-ARRAY` with options as shown above creates an empty string that can expand to accommodate new output. As you can see, formatting additional output to this string appends the new output to whatever is already there. To empty the string, you can either reset its fill pointer (as shown) or create a new emtpy string.

`FORMAT` returns `NIL` except when the destination is `NIL`.

The format control string contains literal text and formatting directives. Directives are always introduced with a ~ character.

Directive	Interpretation
`~%`	new line
`~&`	fresh line
`~|`	page break
`~T`	tab stop
`~<`	justification
`~>`	terminate ~<
`~C`	character
`~(`	case conversion
`~)`	terminate ~(
`~D`	decimal integer
`~B`	binary integer
`~O`	octal integer
`~X`	hexadecimal integer
`~bR`	base-b integer
`~R`	spell an integer
`~P`	plural
`~F`	floating point
`~E`	scientific notation
`~G`	~F or ~E, depending upon magnitude
`~$`	monetary
`~A`	legibly, without escapes
`~S`	READably, with escapes
`~~`	`~`

The first few directives in the table above are for generating whitespace. A `~%` directive inserts a newline character. `~&` inserts a newline only if `FORMAT` output is not already at the beginning of a new line. `~|` generates a page break character; not all devices are capable of responding to this character. You can cause `FORMAT` to emit multiple line or page breaks by using an optional argument, as in `~5%` which generates five newlines.

To advance output to column *n* by inserting spaces, use ~*n*`T`.

You can justify output using `~<` and `~>` to enclose the output to be justified. `~w<`*text*`~>` right-justifies *text* in a field of width *n*. If you segment *text* using `~;` as a separator, the segments will be evenly distributed across the field of width *w*.

`~C` formats a character. Normally, the character formats as itself. However, if you modify the directive to `~:C`, then non-printable characters are spelled out.

```
? (format nil "~:C" 7)  ;; 7 is ASCII BEL
"Bell"
```

You can change the alphabetic case by enclosing output in `~(` and `~)`. Different forms of the directive produce different results. `~(`*text*`~)` converts *text* to lower case. `~:@(`*text*`~)` converts *text* to upper case. `~:(`*text*`~)` capitalizes each word in *text*. `~@(`*text*`~)` capitalizes the first word in *text* and converts the rest to lower case.

The `~D` directive formats an integer as decimal digits with a leading minus sign if the number is negative. `~`*w*`D` right-justifies the output in a field of width *w*, padding on the left with spaces. `~`*w*`,'`*c*`D` pads on the left with the character *c*. Adding the `@` modifier immediately after the `~` causes a leading plus sign to be emitted for positive numbers as well as the leading minus sign that is always emitted for negative numbers. A `:` modifier causes commas to be inserted between each group of three digits.

You can format numbers in binary (base 2), octal (base 8) or hexadecimal (base 16) radix using the directives `~B`, `~O`, and `~X`, respectively. Except for the radix, these behave exactly as `~D`.

The `~R` directive has two forms. `~`*b*`R` prints an integer in base *b*. The directive `~10R` is identical to `~D`. Optional values should be specified following the radix. For example, *~3,8R* prints a base-3 number in a field of width 8.

Without the radix specifier, `~R` spells out an integer as a cardinal (counting) number in English.

```
? (format nil "~R" 7)
"seven"
? (format nil "~R" 376)
"three hundred seventy-six"
```

`~:R` spells out an ordinal (positional) number.

```
? (format nil "~:R" 7)
"seventh"
```

You can even print Roman numerals using `~@R`.

```
? (format nil "~@R" 1999)
"MCMXCIX"
```

One of the hallmarks of professional-looking output is getting plurals right. Lisp makes this easy with the `~P` format directive. Unless its argument is 1, `~P` formats an "s". `~@P` formats a "y" if its argument is 1, or an "ies" if the argument is not 1. Since these operations so commonly follow the printing of a number, `~:P` and `~:@P` reuse the previously consumed argument.

```
? (format nil "~D time~:P, ~D fl~:@P" 1 1)
"1 time, 1 fly"
? (format nil "~D time~:P, ~D fl~:@P" 3 4)
"3 times, 4 flies"
```

You can print floating-point numbers in four different ways:

`~F`

As a number with a decimal point, e.g. 723.0059

`~E`

In scientific notation, e.g. 7.230059E+2

`~G`

As the shorter of the above two representations, followed by a tab to align columns

`~$`

As a monetary value, e.g. 723.01

There are many options for printing floating point numbers. The most common is the field-width specifier, which behaves as it does in the `~D` directive.

Non-numeric Lisp data is printed using the `~A` and `~S` directives. Use `~A` when you want to print data in its most visually attractive form; that is, without the escape characters that would let the data be read correctly by `READ`. Using `~A`, strings are formatted without quotes, symbols are formatted without package prefixes or escapes for mixed-case or blank characters, and characters are printed as themselves. Using `~S`, every Lisp object is printed in such a way that `READ` can reconstruct the object (unless, of course, the object does not have a readable representation).

Iteration

List elements can be formatted using the directive `~{`*format-control*`~}`. The argument *must* be a list; *format-control* consumes elements of the list. Iteration terminates when the list is empty before the next pass over *format-control*. Here's a simple example:

```
? (format t "~&Name~20TExtension~{~&~A~20T~A~}"
          '("Joe" 3215 "Mary" 3246 "Fred" 3222 "Dave" 3232 "Joseph" 3212))
Name                Extension
Joe                 3215
Mary                3246
Fred                3222
Dave                3232
Joseph              3212
NIL
```

If your list might end in the middle of the iteration's *format-control*, you can insert a `~^` directive at that point. If the argument list is empty when interpreting the `~^` directive, the iteration `~{`*format-control*`~}` terminates at that point.

Additional options let you limit the number of iterations and specify different requirements for the arguments. Consult a Lisp reference for details.

Conditionals

Conditional format directives are introduced by `~[` and delimited by `~]`. There are several forms, which I'll call ordinal, binary, and conditional. The ordinal form is `~[`*format-0*`~;`*format-1*`~;`*...*`~;`*format-N*`~]`, which selects the *format-Ith* clause for an argument value of *I*.

```
? (format t "~[Lisp 1.5~;MACLISP~;PSL~;Common Lisp~]" 2)
PSL
NIL
```

Within `~[` and `~]` you can specify a final *default* clause as `~:;`*format-default*; this is selected if the argument is outside the range 0 to N.

The binary form is written `~:[`*format-false*`~;`*format-true*`~]`. The *format-false* clause is interpreted if the argument is `NIL`; otherwise, the *format-true* clause is interpreted.

```
? (format t "My computer ~:[doesn't~;does~] like Lisp." t)
My computer does like Lisp.
NIL
? (format t "My computer ~:[doesn't~;does~] like Lisp." nil)
My computer doesn't like Lisp.
NIL
```

The conditional form, written as `~@[`*format*`~]`, first tests its argument. If the argument is not `NIL`, it is not consumed; rather, it is left for *format* to consume. If the argument is `NIL`, then it is consumed and *format* is not interpreted.

```
? (format nil "~{~@[~A ~]~}" '(1 2 nil 3 t nil 4 nil))
"1 2 3 T 4"
```

Floobydust

As you've seen, many of the format directives accept optional parameters, such as the field width parameter of the justification, tabbing, and numeric directives. Our examples have encoded these parameters into the format control string. Sometimes it is useful to have a parameter vary during the program's operation. You can do this by specifying `V` where the parameter would appear; the parameter's value is then taken from the next argument in the argument list.

```
? (format t "~{~&~VD~}" '(5 37 10 253 15 9847 10 559 5 12))
   37
        253
             9847
        559
   12
NIL
```

In this example, the arguments are consumed in pairs, with the first of each pair specifying a field width and the second being the number to print.

Contents | Cover
Chapter 23 | Chapter 24 | Chapter 25

Copyright © 1995-2001, David B. Lamkins
All Rights Reserved Worldwide

This book may not be reproduced without the written consent of its author. Online distribution is restricted to the author's site.

Chapter 25 - Connecting Lisp to the Real World

Lisp provides a wonderful development environment, as we'll see in the next few chapters. But Lisp would be of little value for some applications without a way to access external programs written in other languages. Fortunately, modern Lisp implementations have a Foreign Function Interface, or FFI for short.

In this chapter I'll describe FFI in general terms. Implementations differ in the details since FFI has not (yet) been standardized. Despite the lack of standardization, current implementations seem to have converged on a similar set of features.

Foreign Function Interfaces let you talk to programs written in "foreign languages"

An FFI lets your Lisp program interact with code that is "foreign" -- i.e. not Lisp.

> This Lisp-centric view of the world is probably motivated by the Lisp machines, where everything -- even the low-level portions of the OS -- was written in Lisp. A good many of the people involved with Lisp during that time are responsible as well for the development of modern Lisp implementations; hence, the not-so-subtle nod toward the notion of Lisp as the center of the programmer's universe.

A typical FFI provides for both calls from Lisp to separately-compiled code, and from separately compiled code to Lisp. (In the latter case, it is almost always true that the external code must have been called from Lisp; it can then call back into Lisp.) Most often, an FFI supports a C calling convention.

Would you wrap this, please?

Why is an FFI even necessary? Why can't you link-in separately compiled code as in any other language? The main reason is that Lisp datatypes don't generally have equivalents in conventional languages. For example, C integers typically fill (depending upon declaration) one-half, one, or two machine words and produce mathematically incorrect results when a result exceeds the representational capacity of the integer's storage. A Lisp integer can fit in a machine word, saving a few bits for a type tag. These are called fixnums. Lisp integers having magnitudes exceeding the capacity of the single word representation are converted to a representation that has an unlimited number of bits -- these are bignums. And with a good compiler, you can define subtypes of integers that, when packed into an array, have just enough bits in their representation to handle the declared range of values.

So, one purpose of an FFI is to translate Lisp datatypes to (and from) "foreign" datatypes. Not all conversions are possible -- a good FFI will signal an error when a conversion is not possible at runtime.

When a non-Lisp function accepts or returns values in a record datatype, the FFI must provide a means of constructing appropriate records. Typically, the FFI gives you a way to construct records that are bit-for-bit identical to those that would have been produced by another language. Fields within a record are set and retrieved using specialized Lisp accessors.

An FFI must also support the proper function calling protocol for non-Lisp functions. Protocols differ by platform and by language. Lisp function calling conventions normally differ from those used by other languages. Lisp supports optional, keyword, default, and rest parameters, multiple return values, closures, and (sometimes, depending upon the compiler) tail call elimination; a conventional language might implement tail call elimination.

What else must an FFI do? It loads object files produced by other languages, providing linker functionality within Lisp for these object files. A linker resolves named entries to code in the object file, and fills in machine addresses in the object code depending upon where the code loads into memory.

Finally, an FFI must resolve differences in memory allocation between Lisp and other languages. Most Lisp implementations allow objects to move during operation; this improves the long-term efficiency of memory management and can improve the performance of the program under virtual memory implementations by reducing the size of the working set. Unfortunately, most other languages expect objects to remain at a fixed memory address during program execution. So the FFI must arrange for a foreign function to see Lisp objects that don't move.

All of the above functionality is encapsulated by an FFI wrapper function. All you have to do is define the name, calling sequence and object code file of some foreign function, and the FFI will generate a wrapper that does all of the necessary translations. Once you've done this, the foreign function can be called just like any other Lisp function.

I'll call you back...

Usually a foreign function is called for its results or to have some effect on the external environment, and a simple call/return sequence is all that's needed. But some foreign functions, particularly those that deal with user interface or device I/O, require access to *callback functions* during their operation. A callback function is called *from* the foreign function.

To define a callback function in Lisp, the FFI basically has to solve all of the foreign function problems in the reverse direction. You use the FFI to define a Lisp function that is callable from another language. The result is typically a function object or pointer that can be passed (as a parameter) to a call of a foreign function. When the foreign function is called, it references the callback parameter in the normal manner to invoke the Lisp callback function. The FFI wrapper around the callback translates the foreign calling sequence and parameter values to the corresponding Lisp format, invokes the Lisp callback function, and returns the callback's results after suitable translation from Lisp to foreign formats.

Network Interfaces: beyond these four walls

Although network protocols are hightly standardized and interoperable, networking APIs are not. Common Lisp vendors usually provide their own interface to the target platform's networking software. Franz's Allegro Common Lisp provides a simple sockets library for IP networking. Digitool's Macintosh Common Lisp comes with a complete set of interfaces to the low-level networking APIs (IP, AppleTalk and PPC) of the Mac OS, plus a collection of sample code that uses the low-level calls to perform common networking tasks; you can use the samples as-is or customize them to your requirements.

Contents | Cover
Chapter 24 | Chapter 25 | Chapter 26

Copyright © 1995-2001, David B. Lamkins
All Rights Reserved Worldwide

This book may not be reproduced without the written consent of its author. Online distribution is restricted to the author's site.

Chapter 26 - Put on a Happy Face: Interface Builders

Graphical user interfaces (GUIs) have changed the way that people use computers. High-end Lisp systems had sophisticated GUIs as far back as the late 1970s, with low-cost consumer computers adopting GUIs in 1984. Modern Lisp systems include tools to build GUIs using both platform-specific and platform-independent techniques. The former can take advantage of proprietary features of the platform's user interface, while the latter provide an abstraction that is portable across multiple platforms.

Event-driven interfaces

Events are key to the operation of all GUIs. An event is a gesture initiated by the user: typically a keystroke, mouse movement or click, menu selection, pen stroke, or speech utterance. An event can occur at any time. This means that the program must be prepared to handle any event at any time in some meaningful way. The interpretation of an event will depend upon the current state of the program, e.g. what windows are visible on the screen and what each window is displaying. An event may change the state of the program and therefore affect the interpretation of later events. But in all cases, the program must be prepared to handle receipt of any event at any time.

Event-driven programs have a control structure known as an *event loop*. The event loop receives events and dispatches them to some part of the program, normally the portion of the program that is in control of the current *focus*, or site of user interest, among all of the information currently displayed by the program.

Graphical programming

The next, and more obvious, characteristic of graphical user interfaces is that they rely entirely upon graphics. Even text is displayed as a graphical image. Of course, it would be incredibly painful (and silly) for every programmer to write programs to render text, lines, circles, boxes, menus, controls, etc. The operating system provides a collection of library routines to draw graphical objects, windows, and controls; the Lisp environment typically provides wrappers (often by use of the Lisp foreign function interface) around the graphics routines so that they may be called from within Lisp.

The availability and implementation details of graphics routines vary widely from platform to platform. You should consult the documentation for your Lisp implementation to learn about how it supports graphics.

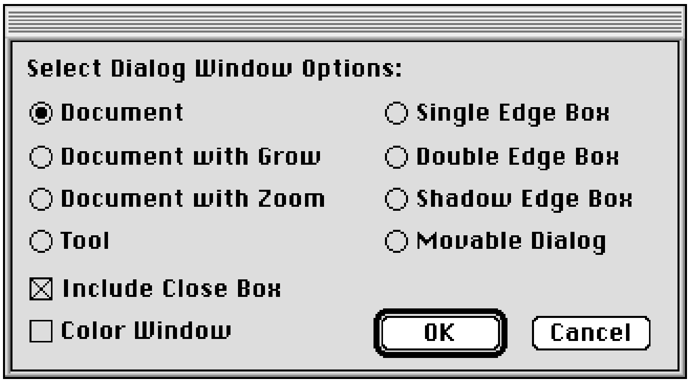
Example: MCL's Interface Toolkit

Macintosh Common Lisp (MCL) provides access to the underlying graphical toolkit of the Macintosh OS in two ways. MCL provides a high-level interface for presenting windows, menus, controls, text, and graphics. This interface is at a higher level of abstraction than the underlying OS primitives; it separates the programmer from concerns about memory allocation, record layout, and pointers. MCL also provides a low-level interface that lets you program with the underlying OS routines (not just for graphics, but for the entire OS). When you use the low-level interface, you are faced with all the concerns that dog a C or Pascal programmer -- only the syntax is different.

In addition to the programmer's interfaces to the Macintosh OS, MCL also provides a tool for visual construction of user interface elements. The Interface Toolkit lets you design windows and dialogs by simply specifying a particular type of new window, then dropping user interface elements into the window. The Interface Toolkit also provides an editor for menus. When you are satisfied with the appearance of your new window or menu, the Interface Toolkit will emit the Lisp code needed to reconstruct it from scratch.

Creating a simple dialog

MCL's Interface Toolkit allows you to create dialogs and menus. To create a dialog, you first select a window style.

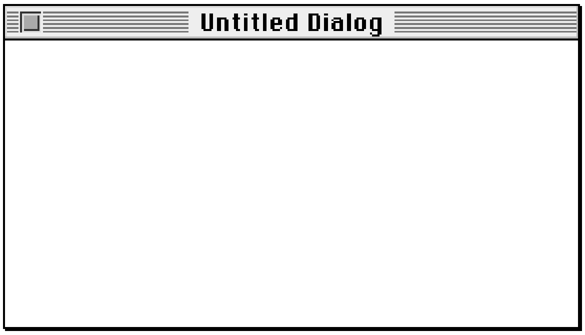


Select Dialog Window Options:

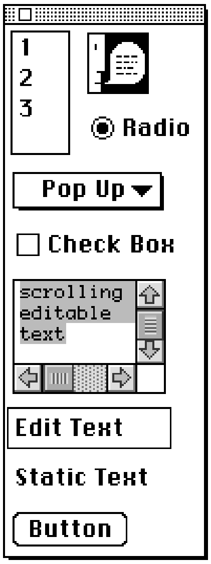
● Document
○ Document with Grow
○ Document with Zoom
○ Tool
○ Single Edge Box
○ Double Edge Box
○ Shadow Edge Box
○ Movable Dialog

☒ Include Close Box
☐ Color Window

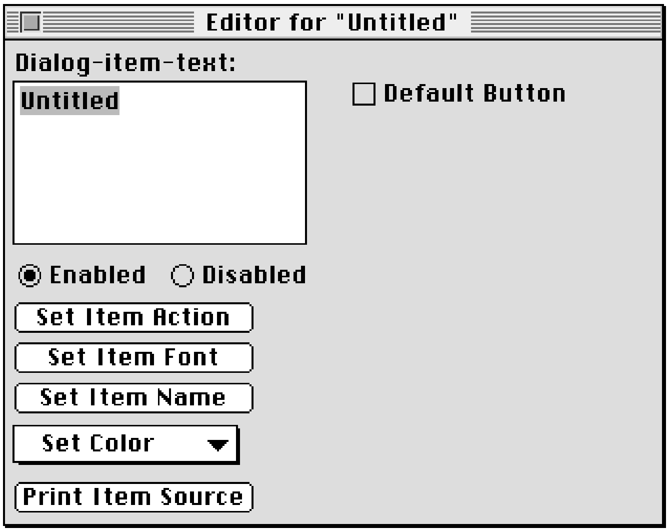
Here, I've chosen to create a simple document window.


Untitled Dialog

A palette of controls appears near the new window. From this palette you can drag and drop controls to create the desired window layout.


1
2
3
Radio
Pop Up
Check Box
scrolling editable text
Edit Text
Static Text
Button

Each control dragged onto the window can be moved and resized. You can also edit attributes of the control, as shown here for a button control.


Editor for "Untitled"

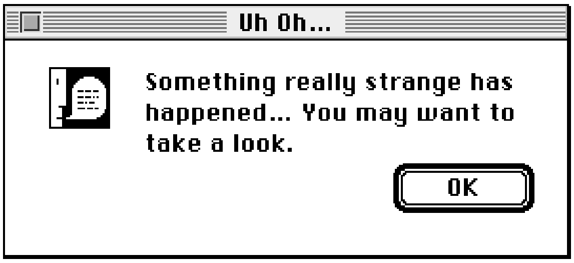
Dialog-item-text: Untitled

☐ Default Button

● Enabled ○ Disabled

Set Item Action
Set Item Font
Set Item Name
Set Color
Print Item Source

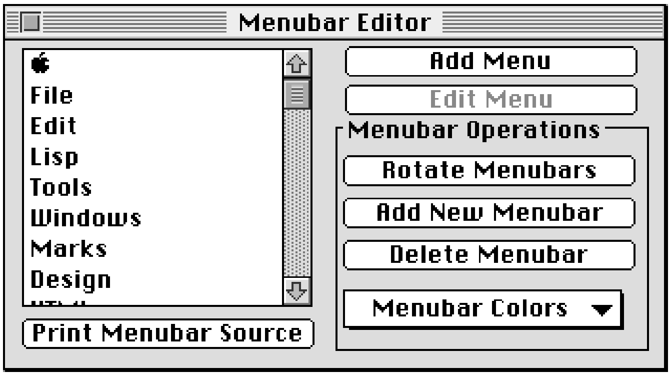
Less than a minute later, I've finished this simple dialog. I can now use a menu command to dump the Lisp source text which will recreate this dialog.


Uh Oh...

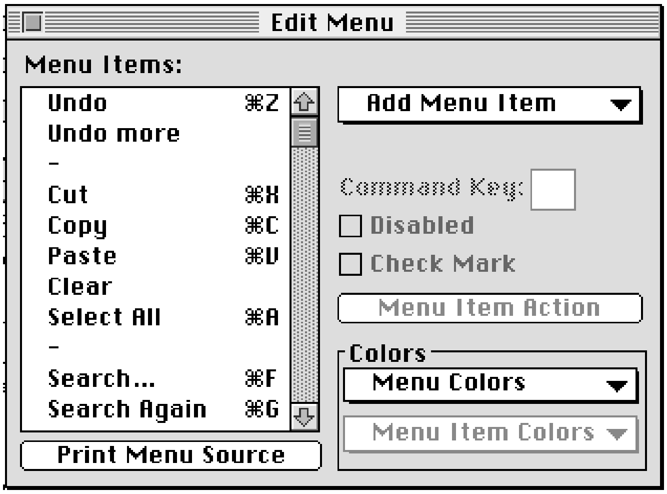
Something really strange has happened... You may want to take a look.

Editing a menu

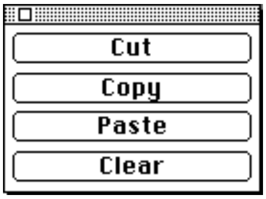
The Interface Toolkit also lets you create and edit menubars, menus, and menu items. You begin with the menubar editor.


Menubar Editor
File
Edit
Lisp
Tools
Windows
Marks
Design
Print Menubar Source
Add Menu
Edit Menu
Menubar Operations
Rotate Menubars
Add New Menubar
Delete Menubar
Menubar Colors

Here, I've chosen to edit MCL's Edit menu.


Edit Menu
Menu Items:
Undo ⌘Z
Undo more
-
Cut ⌘X
Copy ⌘C
Paste ⌘V
Clear
Select All ⌘A
-
Search... ⌘F
Search Again ⌘G
Print Menu Source
Add Menu Item
Command Key:
☐ Disabled
☐ Check Mark
Menu Item Action
Colors
Menu Colors
Menu Item Colors

While you're editing menus, you may not have access to an Edit menu and its Cut, Copy, Paste, and Clear commands. MCL provides a palette with these controls during menu editing.


Cut
Copy
Paste
Clear

Platform-independent interfaces

A platform-independent interface abstracts away details of the underlying operating system's GUI, providing its own event loop, windows, menus, and controls. When you write a GUI using these abstractions, the code can be moved to any other Lisp platform (assuming availability of the platform-independent interface) through recompilation.

CLIM is a commercially-supported platform-independent user interface available on all Lisp platforms. CLIM 2.0 even preserves the native look and feel of each platform by mapping platform-independent requests for windows, menus, and controls onto calls to the native OS graphics services.

Garnet is a free, unsupported platform-independent user interface that is available for most Lisp platforms. Source code is freely available, so you could port Garnet to a new platform if you are so inclined. Unlike CLIM 2.0, Garnet uses its own definitions for windows, menus, and controls; this means that a Garnet GUI will look the same regardless of platform.

Contents | Cover
Chapter 25 | Chapter 26 | Chapter 27

Copyright © 1995-2001, David B. Lamkins
All Rights Reserved Worldwide

This book may not be reproduced without the written consent of its author. Online distribution is restricted to the author's site.

Chapter 27 - A Good Editor is Worth a Thousand Keystrokes

Lisp's simple syntax combines with an integrated editor to ease many of the common tasks of writing a Lisp program. Anyone who tries to tell you that "it's hard to balance parentheses in a Lisp program" is using the wrong editor.

Simple syntax; smart editors

Lisp has a very simple syntax; it's just a bunch of tokens bracketed by a pair of parentheses, recursively. This simple syntax, combined with the fact that the first token following a left parenthesis usually says something about the meaning of the following tokens, lets editors do relatively smart things with program text given only local information.

Virtually every Lisp environment comes with its own Lisp-aware editor. The rare (usually free and minimalist) Lisp environment that doesn't provide its own editor can use Emacs, which has a mode for editing Lisp source code.

Matching and flashing

As you type a Lisp program for the first time, it's handy to see the matching parentheses at a glance. Most Lisp editors facilitate this by briefly highlighting the opening parenthesis for each close parenthesis that you type. Highlighting can take different forms, depending upon the implementation. Sometimes, the text insertion cursor jumps back to the opening parenthesis for a fraction of a second (and then returns to its proper position before inserting the next typed character.) Another common technique is to briefly display the opening parenthesis in a bold font or a different color. And some editors even draw an outline around the entire list. No matter how it's done, you can see at a glance how your closing parentheses match their opening counterparts, with not so much as a pause in your typing of the program.

Once you've entered the program, you can find matching parentheses by positioning the cursor to either the beginning or ending parenthesis, then typing a keystroke that will either flash or move the cursor to the matching parenthesis.

Automatic indentation

Parenthesis matching is important when you're entering or editing a program. When you're reading a Lisp program, proper indentation is important to give you visual cues as to the program structure. In fact, you should be able to hide the parentheses in a properly indented Lisp program and still understand the program.

Lisp editors typically supply proper indentation as you type a new program; with some editors this is done automatically, while others require that you use a different keystroke in place of the return key to end a line. Together with parenthesis matching, automatic indentation lets you type a properly parenthesized and indented Lisp program without ever having to count parentheses or spaces.

When you edit an existing Lisp program, you'll often add or remove portions such that the proper indentation of the rest of the program must change. If you had to readjust everything by hand, you'd become quickly disenchanted with Lisp. That's why Lisp editors give you a way to reindent a program either a line at a time, or for an entire Lisp form. Lisp programmers tend to develop a habit of finishing small changes with the keystroke to reindent the portion of the program being edited; this gives the programmer immediate visual feedback on the updated structure of the program, as expressed by its indentation.

Symbol completion

Lisp programmers tend not to abbreviate names. The short, mnemonic names in Common Lisp are there for historical reasons; the newer additions to Lisp have fully spelled out, descriptive names. Fortunately, a good Lisp program editor can save you a lot of typing by providing a symbol completion facility.

Symbol completion works like this. As you type your program, you're using names that are both built into the Lisp system and defined anew by your program. As you type a name, you can press a symbol-completions keystroke anytime after having typed the first few characters. If the typed prefix *uniquely* matches some symbol already known to the Lisp environment (either because it's built in, or because you've previously typed the whole name), the editor will type the rest of the symbol for you. If the typed prefix matches more than one completion, the editor may either pick the first and let you cycle through the rest by repeating the completion keystroke, or it may present a list of possible completions from which you can choose. In any case, it's important to note that symbol completion does *not* depend upon having compiled your program; completion works even during initial program entry.

Finding definitions

As you develop your program, you'll often find that it's helpful to refer to a function that you've defined earlier; perhaps you need to see the source code to confirm that it will respond appropriately to unexpected inputs, or maybe you need to see how some vendor-supplied function is implemented. Lisp editors support this kind of exploration with yet another keystroke; just position the cursor somewhere in a name, press a key, and (if the definition is accessible in source form) you're instantly shown a view of the defining source code.

On-line documentation

Despite its underlying simplicity, Lisp is a large language. The formal ANSI specification fills some 1,500 pages of paper describing the language and 978 predefined symbols. On top of that, your vendor's development environment will define hundreds of thousands of additional symbols. And, of course, your program will only add to the roster of symbol names.

Most Lisp editors will let you access documentation in various forms. For built-in and user-defined code, the editor should give you quick access to the documentation string and argument list. In fact, many Lisp editors automatically find and inobtrusively display the argument list whenever you type a space following a function or macro name. A quick glance at the argument list helps you avoid usage mistakes.

Hypertext access to online manuals is becoming increasingly popular. Some Lisp editors will support lookups in one or more manuals (including the Lisp reference) in a manner similar to the way they support access to documentation strings.

Access to debugging tools

The editor (or the Lisp environment in general if the environment is not editor-centric) should support easy access to debugging tools such as browsers, trace and step facilities, backtraces, inspectors, and program databases such as apropos, who-calls, and callers-of (see Chapters 10, 16, and 22).

Extending the editor using Lisp

The editor in many Lisp development environments is itself written in Lisp. The vendor should document the external APIs of the editor and supply source code; then you can add your own extensions to the editor and customize your Lisp environment to fit the way that you work.

Contents | Cover
Chapter 26 | Chapter 27 | Chapter 28

Copyright © 1995-2001, David B. Lamkins
All Rights Reserved Worldwide

This book may not be reproduced without the written consent of its author. Online distribution is restricted to the author's site.

Chapter 28 - Practical Techniques for Programming

In this chapter, we'll learn some brief yet useful guidelines for Lisp style, followed by practical advice on tradeoffs among debugging, performance, and readability.

Elements of Lisp style

The art of Lisp style is simpler and more fruitful than in most other languages. Lisp's simple, consistent syntax eliminates the need for the rules of style that plague more complicated languages. And the direct, standardized availability of complex functionality within Lisp helps to drive down the size of programs, thereby providing improved readability through brevity. (In my experience, a Lisp program may range in size from 5 percent to 20 percent of an equivalent C++ program.)

Furthermore, the universal availability of Lisp-aware program editing tools -- such as Emacs or its equivalent built into many Lisp IDEs -- means that you can let the computer handle the details of indentation that are so important to the ability of a person to comprehend the structure of a Lisp program. I've said this before, but it bears repeating: you should *not* be programming in Lisp without the aid of a Lisp-aware editor.

So, if we don't have to worry about conventions for spelling, capitalization, indentation, and other such mundane details, then what remains for us to discuss as elements of Lisp style? How about things that are truly important? Lisp programming style is about the choice of proper abstractions, and about communicating not just with the compiler, but with other people who will eventually read your program. (For that matter, good style will help you read your own program some months or years in the future.)

Property lists are handy for small (very small) ad-hoc databases

A long, long time ago the capabilities of a typical Lisp implementation were very much less than what you'll find in any Common Lisp system today. After all, Lisp has been around for over forty years since John McCarthy first invented the notations (see Chapter 34). Strangely, when Lisp is taught at all in computer science curricula, it is taught using a circa-1965 view of the state of Lisp implementations: interpreted execution, limited data structures, and no real application beyond the manipulation of symbols.

Unfortunately, authors and publishers of Lisp textbooks did little to help correct these misperceptions, ignoring Common Lisp (and indeed, many of its recent forebears) in highly-recommened Lisp textbooks published as recently as 1989.

In the bad old days -- when Lisp didn't have arrays, vectors, hash tables, structures, or CLOS -- programmers learned to rely heavily on property lists as an important mechanism for structuring data. You'll still find -- in bookstores and on the shelves of college libraries -- Lisp and AI books that recommend the use of property lists as the underlying basis for looking up values identified by a symbolic key.

A property list is a list of alternating keys and values. For example: the list `(SEX MALE PARENTS (BOB JANE) OCCUPATION MUSICIAN)` establishes these relations:

```
relation   value
---------- ----------
sex        male
parents    (bob jane)
occupation musician
```

When you attach these relations to a symbol, say `JAMES`, then you have a way to get information related to the symbol `JAMES`, namely the properties having the names `SEX`, `PARENTS`, and `OCCUPATION`.

In Common Lisp you can retrieve a symbol's property using `(GET` *symbol property-name*`)`, and set a property using `(SETF (GET` *symbol* `property-name)` *property-value*`)`.

While property lists were useful substitutes for more capable data structures in ancient Lisp implementations, they find few uses in modern Lisp programs. One problem is efficiency. Every time you ask Lisp to retrieve a property value, it must locate the symbol's property list and then search the list for a matching key. If you have five or six properties on a symbol, the search may or may not be faster than using a hash table; the exact crossover point will depend upon your particular Lisp implementation. The other problem is that properties are shared among all parts of your program. It's not too difficult to imagine two programmers using a property named `COLOR` in two different ways in two parts of the same program. Imagine their surprise when they discover the conflict...

At any rate, you should become familiar with all of the capabilities of Common Lisp, and learn to recognize that information about older Lisp implementation which still haunts us through the mists of history.

Declarations help the compiler, sometimes

Common Lisp defines the following declarations:

special
    declares a variable to have dynamic (not lexical) scope
optimize
    instructs the compiler how to weight the relative importance of
- ◊ speed
- ◊ safety
- ◊ space
- ◊ debug
- ◊ compilation-speed

dynamic-extent
    declares that the programmer expects the lifetime of a function or variable to end when control leaves the enclosing form
type
    declares that a variable will always have values of a given type
ftype
    declares that a function should expect arguments of specified types, and that the function will return values or given types
ignore
    declares that a variable is not referenced
ignorable
    declares that a variable may not be referenced
inline
    declares that the programmer would like a function to be compiled as inline code
notinline
    declares that the programmer does not want a function to be compiled as inline code

Of these, only the first and last *must* be implemented. The rest are advisory; depending upon the implementation, the compiler may or may not honor the given advice. If you've programmed in other languages, you may find it strange that most of Lisp's declarations are advisory rather than mandatory. So let's dig a bit deeper and see what this really means to you.

Lisp by default must have the *capability* to determine the type of every variable at runtime. This is not to say that a sufficiently smart compiler can't infer at compile time that a variable will always be of a particular type and generate code that does not need to check types at run time. However, an actual "sufficiently smart compiler" remains an elusive creature, much like the Yeti, Bigfoot and the Loch Ness Monster.

Declarations allow the programmer to pass metainformation to the compiler. This is not part of the program, but rather information *about* the program. Declarations can help the compiler to generate better code by providing information about the programmer's *intent*.

For example, if you declare that a variable will always be a `FIXNUM` (an integer value that fits in a single machine word) then the compiler can emit code to load that variable directly into a register in preparation for the next operation. If you declare the result of the operation to also be a `FIXNUM`, then the compiler can generate code to perform the operation and store the result using simple machine instructions without first checking the `actual` type of the value. Given such declarations, a good Lisp compiler can generate code comparable to a low-level language in which operations and types in the language map directly onto the underlying machine.

But there's a risk. If you declare certain types, and the compiler emits code that optimizes the program according to your declarations, and the program then *contradicts* those declarations by providing a value of a different type at runtime, then bad things will happen. Tell the compiler to expect two numbers to add, then pass it a number and a symbol, and all bets are off.

Fortunately, the declarations that guide the compiler are themselves moderated by the `OPTIMIZE` declaration. The `OPTIMIZE` declaration lets you instruct the compiler about the relative importance of certain properties of the program. You can specify the relative importance of the `SPEED`, `SPACE`, and `SIZE` of the generated code. You can specify whether you'd like to allow the compiler to spend extra time doing a better job, or to emphasize `COMPILATION-SPEED`. You can specify the importance of being able to `DEBUG` your program, which may cause the compiler to produce code that is simpler or interacts well with the debugger.

> Values range from 0 to 3 for the `OPTIMIZE` declarations, with 0 meaning "totally unimportant" and 3 meaning "most important". The default value is 1, meaning "of normal importance". Bear in mind that for something to be relatively more important, something else must be less important; it won't give the compiler any useful guidance to specify values of 3 for all of the `OPTIMIZE` declarations.

Of all the `OPTIMIZE` declarations, the most important is `SAFETY`, since this affects the amount of trust the compiler is willing to extend to your type declarations. A high value of `SAFETY` generally compels the compiler to check the type of every value that it can't absolutely determine at compile time. Lower `SAFETY` values put increasing weight upon your abilities as a programmer to guarantee that type declarations are correct, array bounds are always in range, etc.

The exact effect of declarations (with the exception of `SPECIAL` and `NOTINLINE`) varies among Lisp implementations; consult your reference manual for details.

DEFVAR versus DEFPARAMETER

Although not required by the Common Lisp standard, almost all implementations require that you load code from a file. (The one exception that I know of is the Venue Medley environment, which normally saves the entire Lisp world when you end a session. Medley also keeps track of new definitions created in the listener and allows you to save just those definitions to a file.)

In a file-based Lisp environment, you'll normally add definitions to a file of source code. One reason for so doing is to periodically save your work; unless you're debugging FFI code or running buggy Lisp code with a low optimization value for safety, your Lisp environment will almost never crash. However, other disasters can happen -- another application could crash and bring down the system in an unprotected OS such as the Mac OS or Windows, the power could fail, or your cat could walk across the keyboard when you leave to refill your coffee.

As your program gets larger, you may find that it's useful to reload an entire source file after making a series of changes. Most Lisp environments also let you evaluate one definition at a time in any open window. This is quite useful because you can edit, then recompile, one definition at a time. But sometimes you'll forget, and then it's easier to just reload the entire file than to spend time figuring out which definition you might have forgotten to recompile after last changing its definition.

But you may also be in the midst of debugging your program when you'd like to reload its source code. If your program uses any global variables to keep track of its state, you really *don't* want to reinitialize these in the midst of your debugging session. So, how do you handle this? You could put definitions of your program's state variables in a separate file, but that increases your mental workload and increases debugging time by splitting clearly interrelated parts of your program into two separate files. (I know that this is an accepted practice in many programming languages, but it really does increase the amount of work you do as a programmer. Imagine how much less pleasurable reading a novel would be if the novel was delivered as a set of pamphlets, one per character, and you had to follow page references to get to the next part of the dialog.)

Fortunately, Lisp has grown through decades of real-world programming experience, and has a very simple mechanism to handle whether variables get reinitialized or not when you load a file. You use `DEFVAR` to declare variables with values that need to be initialized only once. To declare a variable with an initial value that gets set every time its defining form is evaluated, use `DEFPARAMETER`.

One final note: as a matter of form, you should name global variables with a leading and trailing asterisk, as in `*MY-ADDRESS*`. Think of this convention as a courtesy to those who will maintain your code at some future date.

Define constants with DEFCONSTANT

You should define global constants using `DEFCONSTANT`. From the viewpoint of reading a Lisp program, the distinction between `DEFPARAMETER` and `DEFCONSTANT` is that the value defined by `DEFPARAMETER` could concievably be altered by the user after the program is compiled, but a `DEFCONSTANT` value will *never* change. A good Lisp compiler will take advantage of `DEFCONSTANT` declarations to perform classical optimizations such as constant folding or compiling immediate load instructions.

Fewer Lisp programmers follow a naming convention for constants. The one I use puts a leading and trailing plus sign on the name of the constant, as in `+RTS-OPCODE+`.

Know when (not) to use the compiler

Most Lisp systems include both an interpreter and a compiler; when both are available, you'll normally find that it's easier to debug interpreted code. Consult your vendor's documentation to learn how to switch between the interpreter and the compiler.

Of course, when performance is important, you'll want to run your code compiled once it's debugged. But see the earlier cautions about running buggy code with low safety settings.

When you're writing Lisp code to run on multiple platforms, it's safest to assume that code will run interpreted unless you call `COMPILE` or `COMPILE-FILE`. For this reason, you should develop the practice of writing (or using) a system definition procedure that first loads all of your Lisp source files, then compiles them, then loads the compiled files. This is usually overkill, but it's a very safe, conservative approach. With suitable source code organization and proper use of `EVAL-WHEN` you can reduce the number of source files that must first be loaded; the main idea is to ensure that all macros are defined before compiling code that uses the macros, but there are other possible situations that can depend upon the current state of the Lisp world.

Speed vs. ability to debug

Interpreted programs are easier to debug because it's easier for the debugger to access the actual source code at the point of an error. Once you've compiled your program, the debugger typically has less source information available; you may find yourself puzzling over a transformed version of the source code or grovelling through assembly-language instructions to find the source of the error. Fortunately, the need for such low-level debugging will be rare if you follow some simple advice:

1. Keep high `SAFETY` optimizations on untested code.
2. If an interpreter is available to you, use it until your code is working well.
3. If you have a compile-only environment, use lower optimization settings for `SPEED` and higher settings for `DEBUG`.

Once your code is running well, then you should compile it and adjust the optimization declarations for performance. If you find that simply compiling your program provides adequate performance, leave it alone. If the performance of the compiled program falls far below your expectations, first improve the algorithm; optimization declarations typically have a fractional impact upon performance.

Efficiency: spotting it, testing it

The first rule of efficiency in any programming language is to start with an efficient algorithm. It's a little harder to spot inefficiencies in a Lisp program because the underlying operations don't usually map directly onto a hardware instruction. But with a certain amount of knowledge and practice, you should be able to tell why the following four programs have radically different resource requirements.

These four programs return the sum of a list of numbers, but do it in different ways. In each case, we test the program with the `TIME` form, which reports run time and memory allocation. Each program is tested twice, once with a list of ten thousand elements, then again with one hundred thousand.

```
;; Runtime increases as the square of the number of elements
? (defun sum-list-bad-1 (list)
    (let ((result 0))
      (dotimes (i (length list))
        (incf result (elt list i)))
      result))
SUM-LIST-BAD-1
? (let ((list (make-list 10000 :initial-element 1)))
    (time (sum-list-bad-1 list)))
(SUM-LIST-BAD-1 LIST) took 2,199 milliseconds (2.199 seconds) to run.
Of that, 102 milliseconds (0.102 seconds) were spent in The Cooperative Multitasking Experience.
 16 bytes of memory allocated.
10000
? (let ((list (make-list 100000 :initial-element 1)))
    (time (sum-list-bad-1 list)))
(SUM-LIST-BAD-1 LIST) took 336,650 milliseconds (336.650 seconds) to run.
Of that, 15,680 milliseconds (15.680 seconds) were spent in The Cooperative Multitasking Experience.
 2,704 bytes of memory allocated.
100000

;; Recursive version works when compiler does tail-call optimization
? (defun sum-list-bad-2 (list)
    (labels ((do-sum (rest-list sum)
               (if (null rest-list)
                 sum
                 (do-sum (rest rest-list) (+ sum (first rest-list))))))
      (do-sum list 0)))
SUM-LIST-BAD-2
? (let ((list (make-list 10000 :initial-element 1)))
    (time (sum-list-bad-2 list)))
(SUM-LIST-BAD-2 LIST) took 2 milliseconds (0.002 seconds) to run.
10000
? (let ((list (make-list 100000 :initial-element 1)))
    (time (sum-list-bad-2 list)))
(SUM-LIST-BAD-2 LIST) took 21 milliseconds (0.021 seconds) to run.
100000

;; The recursive version can fail w/o tail-call optimization
? (defun sum-list-bad-3 (list)
    (declare (optimize (debug 3)))  ; disable tail-call optimization
    (labels ((do-sum (rest-list sum)
               (if (null rest-list)
                 sum
                 (do-sum (rest rest-list) (+ sum (first rest-list))))))
      (do-sum list 0)))
SUM-LIST-BAD-3
? (let ((list (make-list 10000 :initial-element 1)))
    (time (sum-list-bad-3 list)))
> Error: Stack overflow on control stack.

;; The iterative version is not as elegant, but it's fast!
? (defun sum-list-good (list)
    (let ((sum 0))
      (do ((list list (rest list)))
          ((endp list) sum)
        (incf sum (first list)))))
SUM-LIST-GOOD
? (let ((list (make-list 10000 :initial-element 1)))
    (time (sum-list-good list)))
(SUM-LIST-GOOD LIST) took 1 milliseconds (0.001 seconds) to run.
10000
? (let ((list (make-list 100000 :initial-element 1)))
    (time (sum-list-good list)))
(SUM-LIST-GOOD LIST) took 10 milliseconds (0.010 seconds) to run.
100000
```

The first version, `SUM-LIST-BAD-1`, harbors a hidden inefficiency: `(ELT LIST I)` must search `LIST` from the beginning for each value of `I`. In other words, `ELT` must examine one element when `I` is 1, two elements when `I` is 2, and so on. For a list of length *N*, `ELT` will examine almost *N*-squared elements. Have a look at the runtimes for 1,000 and 10,000 elements.

The second version is coded using an awareness of how lists are accessed; the helper function `DO-SUM` calls itself recursively with the tail of the list it is given. In `SUM-LIST-BAD-2`, the runtime increases linearly with the length of the input list. So why is this a **bad** example?

`DO-SUM` calls itself as the last form it executes; this is known as *tail recursion*. Some compilers can compile tail recursion as a jump instruction instead of a function call; this eliminates the growth of the control (function return) stack that would otherwise occur in a recursive call. However, the Common Lisp standard does not *require* that tail calls be optimized.

The third version shows what can happen when the compiler does not optimize tail recursion. The compiler in my Lisp system disables tail recursion optimizations when the `DEBUG` optimization is set to 3. Without tail recursion optimization, `SUM-LIST-BAD-3` consumes a function call frame for each recursive call, and the program fails -- exhausting stack space -- before reaching the end of the test list.

The final version, `SUM-LIST-GOOD`, uses iteration instead of recursion for its control loop, and walks down the input list element by element. It runs slightly faster than `SUM-LIST-BAD-2` and doesn't fail if the compiler doesn't support tail recursion optimization.

Recognizing inefficiency, profiling; performance vs. readability

Avoidance is the best defense against inefficiency. Use appropriate data structures and control techniques. When you're not sure, put together a *small* test program and time it with a variety of inputs.

Every Common Lisp implementation will have the `TIME` macro that we used to show the differences in the `SUM-LIST-`*xxx* functions. You can use this to examine and tune small portions of a program.

Once you have assembled a larger program, you may need to find the bottleneck that causes unexpectedly low performance. For this, you'll need a profiler. A profiler watches the execution of your whole program and generates a report to show where the program spends its time. Some profilers also report on memory allocation. A profiler is not a standard part of Lisp, but most vendors provide one. Consult your vendor's documentation.

Lisp provides abstractions that help you solve problems. You'll find that you don't have to make a tradeoff between readability and performance; an efficient Lisp program is usually one that is written using the most appropriate abstractions and operations to solve a given problem.

Contents | Cover
Chapter 27 | Chapter 28 | Chapter 29

Copyright © 1995-2001, David B. Lamkins
All Rights Reserved Worldwide

This book may not be reproduced without the written consent of its author. Online distribution is restricted to the author's site.

Chapter 29 - I Thought it was Your Turn to Take Out the Garbage

Chapter objective: Describe the benefits and costs of garbage collection. Show how to improve program performance by reducing the amount of garbage it generates.

What is garbage?

In simplest terms, garbage is any storage that your program once used, but uses no longer. Here's a simple example:

```
(let ((x (list 1 2 3 4 5)))
  (print x))
```

When you evaluate this form, the list `'(1 2 3 4 5 6)` is first bound to `X` and then printed. Once control leaves the `LET` form, the list bound to `X` is no longer accessible; its storage can be reclaimed by the garbage collector.

> Actually, there's a minor complication that you should know about. When you evaluate a form in the Lisp listener, the form itself is assigned to the symbol `+`, and the value is assigned to the symbol `*`. The previous form and value are assigned to `++` and `**`, respectively, and the form and value before that are assigned to `+++` and `***`. Because these three pairs of variables give you a way to access the forms and results, a form and its result can't really become garbage until you've evaluated additional forms to flush these six variables.
>
> You won't normally have to worry about this unless you've done something in the listener to exhaust all available memory in Lisp; if you can evaluate a simple expression (like `T`) three times, you'll release any storage held by `+`, `*`, and friends.

Why is garbage collection important?

Lisp allocates storage as needed for your program's data. You don't have direct control over how or when storage is allocated; the compiler is free to do the best job it can to satisfy the meaning of your program.

Lisp does not provide a way for your program to explicitly deallocate storage. This is an important feature, because you can never write a program to mistakenly deallocate storage that is still needed elsewhere in the program. This eliminates an entire class of errors, sometimes referred to as "dead pointer bugs" in languages that support explicit storage allocation and deallocation.

On the other hand, your program may eventually run out of memory if your program never deallocates storage. So a language (like Lisp) that doesn't support explicit deallocation must still provide a mechanism to automatically deallocate storage when the storage is no longer needed. The garbage collector's job is to figure out which storage can no longer be accessed by your program, and then recycle those inaccessible storage blocks for later use.

How does garbage collection work?

Lisp compiles your program in such a way that all of its allocated storage can be found by following pointers from a small number of known *root* pointers. The compiler and runtime system arrange for your program to retain type information at runtime; this is combined with compile-time knowledge of storage layouts to encode knowledge of the locations of pointers within data structures.

The garbage collector follows every pointer in every reachable data structure, starting with the root set. As it does so, it marks the reachable data structures. Once every pointer has been followed, and its referenced data structure marked, any block of memory that is unmarked is unreachable by your program. The garbage collector then reclaims these unmarked blocks for future use by the storage allocator.

> The actual marking algorithm used by the garbage collector must account for cycles in the reachable data structures, and must perform in limited space and time; these details complicate the implementation of a garbage collector. Also, most collectors will relocate the marked data (and adjust references accordingly). [Jones96] provides an excellent survey and analysis of various garbage collection techniques.

What effect does garbage have on my program?

Garbage causes your program to run slower. The more garbage your program creates, the more time the garbage collector will need to spend recycling the garbage. Modern garbage collectors are very efficient; it's unlikely that you'll see a noticeable pause in your program's execution as the garbage collector runs. However, the cumulative effect of many small pauses will cause a detectable degradation in overall performance.

The good news is that garbage collection ensures that your program will *never* suffer from memory leaks or dead pointers.

Also, because many garbage collector implementations rearrange storage as your program runs, heap fragmentation is minimized; thus, a large Lisp program's performance will not degrade over time like a C or C++ program that performs comparable storage allocation (typically 25 to 50 percent degradation for a C or C++ program, depending upon heap size, malloc/free implementation, and allocation/deallocation patterns).

> You should note that explicit storage allocation and deallocation has overheads which are not strictly predictable. In typical malloc and free implementations, block allocation involves a search and deallocation involves extra work to coalesce free blocks; both of these activities are of effectively indeterminate duration, affected by the size and fragmentation of the heap.

How can I reduce garbage collection pauses in my program?

You can reduce garbage collection overhead by reducing garbage generation. Use appropriate data structures; list manipulation is the most common cause of garbage creation in poorly-written Lisp programs. Pay attention to whether an operation returns a fresh copy or a (possibly modified) existing copy of data.

If you have a profiler available in your Lisp system, use it to find your program's hot spots for storage allocation.

Use destructive operations carefully; they can reduce garbage generation, but will cause subtle bugs if the destructively-modified data is shared by another part of your program.

Contents | Cover
Chapter 28 | Chapter 29 | Chapter 30

Copyright © 1995-2001, David B. Lamkins
All Rights Reserved Worldwide

This book may not be reproduced without the written consent of its author. Online distribution is restricted to the author's site.

Chapter 30 - Helpful Hints for Debugging and Bug-Proofing

As with any programming language, error avoidance is the best debugging strategy. Take advantage of the online documentation (available with most systems) and test functions, or even parts of functions, as you write them.

Still, you'll inevitably face the unexpected error, and you should know how to use the debugger. More often than not, a quick look at the error location as shown by the debugger will point out an obvious problem.

Some problems, though, are not obvious; your program will run without error, but produce incorrect results. When examination of the code does not reveal an error, you can rely upon built in Lisp tools to expose the details of your program's operation and find the error during execution.

Finding the cause of an error

There are two ways to notice an error. The intrusion of the Lisp debugger is the most obvious. The debugger will appear whenever your program causes Lisp to signal an error. This is often the result of something obvious, like trying to perform arithmetic on `NIL` or trying to `FUNCALL` an object that is not a function.

Your program's failure to produce expected results is also an error, even though the debugger never appears. In this case, your program doesn't make any mistakes in its use of Lisp, but the successful sequence of Lisp operations doesn't do what you had intended. Another possibility is that your program will fail to terminate at all.

The best defense against all of these problems is to write short, clear function definitions and test each one as soon as you've written it. I find it helpful to write one or more test cases and include them as comments (bracketed by #| and |#) in the same file.

Reading backtraces, compiler settings for debugging

Every Lisp debugger will provide at least two important pieces of information: an error message and a stack backtrace.

The error message describes *how* the program failed. Normally, this is a description of an error encountered while executing some built in Lisp function. If your program calls *ERROR*, the debugger will display the message you specify.

The stack backtrace describes *where* your program failed by displaying the call stack at the point of the error. The function which signalled the error will be at the "top" of the stack. Below that is the function that called the function which signalled the error, and so on all the way to (and sometimes beyond) the listener's read-eval-print loop.

The debugger relies upon certain information provided by the compiler or interpreter. Although the details vary among implementations, it's safe to say that compiler optimizations that improve speed or reduce space tend to reduce the amount of information available to the debugger. You can change these optimizations while debugging your program:

```
(declaim (optimize (speed 0) (space 0) (debug 3)))
```

If you execute this before compiling your program, the debugger should be able to give you more useful information. You should consult your vendor's documentation to learn about additional implementation-specific controls. If your Lisp system gives you a choice between using a compiler and using an interpreter, you'll probably find that the interpreter causes the debugger to give you better information.

Simple debugging tools

If your program runs to completion but produces incorrect results, or if it runs but fails to terminate, then you'll need some additional tools. The first of these tools should be familiar to all programmers: insert a call to the debugger or (more commonly) insert a print statement.

BREAK, PRINT

`BREAK` causes your program to call the debugger. Once inside the debugger you can examine the call stack. Most debuggers also allow you to examine values local to each active function on the call stack; by looking at these values at a critical point during your program's execution, you may find an important clue as to why your program malfunctions.

The debugger will allow you to continue from a break. You may find it helpful -- if you don't yet understand the cause of a problem -- to correct one or more wrong values before continuing; with other `BREAK` forms inserted at key points in your program, this strategy may lead you to a place where the error *is* apparent.

Of course, you can always insert `PRINT` forms at key locations in your program and examine the resulting output. In Lisp, this is most useful when you need to get a feel for what's happening deep inside some function. For example, you might have a complex calculation to determine whether a sequence of code is executed or not. A `PRINT` can tell you as the program runs.

Don't forget that you can use `FORMAT` to print the values of several variables together with explanatory text. And with either `PRINT` or `FORMAT`, be careful that you do not change the meaning of the code by inserting the debugging statement. Remember that some flow-control forms (e.g. `IF` and `UNWIND-PROTECT`) expect a single form at certain places. Also beware of wrapping `PRINT` around a value-returning form; this won't work if the value-receiving form expects multiple values.

Power tools for tough problems

Lisp provides additional debugging tools to help you observe the dynamic behavior of your program.

TRACE, STEP, ADVISE, WATCH

`TRACE` allows you to observe each call and return from a specific function, no matter where the function appears in your program. To trace a function, invoke `TRACE` with the name of the function. You can do this for as many functions as needed. You can also pass several function names to `TRACE`.

When your program runs a traced function, it will print the name of the function on entry and exit. Most `TRACE` implementations will also print the function arguments on entry and returned values on exit.

To discontinue tracing of a function, pass its name to `UNTRACE`. To discontinue tracing of *all* traced functions, evaluate `(UNTRACE)`.

See Chapter 16 for an example of `TRACE`.

`STEP` allows you to interactively control evaluation of an expression. If you step a function invocation, you should be able to examine each subform of the function's definition just before it is evaluated. `STEP` implementations vary widely, so you should consult your vendor's documentation for further details. In general, the same optimizations and controls that aid the debugger will also aid the stepper.

`STEP` is a very labor-intensive way to debug a program, since you must tell its user interface to evaluate each subform. This is reasonable for straight-line code, but quickly becomes tedious in the presence of looping or recursion.

Some Lisp implementations provide two additional tools, `ADVISE` and `WATCH`, that can be of use during debugging.

`ADVISE` modifies a function without changing its source code. `ADVISE` can usually examine the advised function's arguments, execute its own code, execute the advised function, examine the advised function's return values, and modify the returned values. For debugging purposes, `ADVISE` can be used to implement conditional `BREAK`s and `TRACE`s, or to temporarily patch incorrect behavior in one part of a program while you're debugging another part.

`WATCH` lets you specify variables to be displayed as your program executes. This is normally available only in Lisp implementations that provide a windowed user interface. Because of issues of variable scope and display update timing and overhead, `WATCH` is of limited value. Most Lisp implementations do not provide this tool.

Into the belly of the beast

As you debug your program, you may need to see the internal details of composite objects such as lists, structures, arrays, streams and CLOS instances. Lisp lets you do this whether the data has been defined by your program or by the Lisp runtime system.

INSPECT, DESCRIBE

`DESCRIBE` is a function that accepts any object as an argument and prints a description of that object. The form and content of the description may vary among Lisp implementations. `DESCRIBE` accepts an output stream as an optional second argument.

`INSPECT` is an interactive version of `DESCRIBE`. This is most useful for examining complex objects by "drilling down" into the implementation details of enclosed data elements.

Continuing from an error

When faced with the debugger, you will have a choice of restart actions depending upon how the error was signalled. `ERROR` requires that you abandon your program's executions. However, many internal Lisp functions use `CERROR`, which gives you a chance to continue from an error.

In most debuggers, you can do quite a few useful things before continuing from an error:

- alter variable values
- redefine the function that caused the error and run it again

- skip the rest of the function that caused the error and specify values to be returned
- restart any function further down the call stack
- skip the rest of any function further down the call stack and specify values to be returned

Problems with unwanted definitions

Unwanted definitions are not usually a problem in a Lisp program. You can get rid of function definitions using `FMAKUNBOUND`, variable values with `MAKUNBOUND`, and even symbols with `UNINTERN`. In practice, there's usually no need to use any of these; available memory is commonly large compared to the size of a few misdefined variables or functions, and they will be eliminated anyway the next time you restart your Lisp image and load your program.

Method definitions are an entirely different matter. Remember that methods must have congruent argument lists; if you change your mind during program development about a method's argument list -- perhaps you thought that it needed two arguments at first but then realized three arguments are really needed -- then you'll have to remove the old method definition before adding the new one. Some Lisp environments facilitate this redefinition:

```
? (defmethod baz (a b))
#<STANDARD-METHOD BAZ (T T)>
? (defmethod baz (a b c))
Error: Incompatible lambda list in #<STANDARD-METHOD BAZ (T T T)>
       for #<STANDARD-GENERIC-FUNCTION BAZ #x3D2CB66>.
Restart options:
  1. Remove 1 method from the generic-function and change its lambda list
  2. Top levl
  ?
```

If you simply add a method that you later decide is no longer wanted, you'll need a way to remove the method. The least desirable technique is to restart your Lisp system and reload your program without the unwanted definition. Another approach, provided by some vendors, is to interactively remove the definition using a special editor command or a method browser. Failing all else, you can remove the method manually using `REMOVE-METHOD`:

```
(let* ((generic-function (symbol-function 'gf-name))
       (method (find-method generic-function
                            '(method-specializers)
                            (list (find-class parameter-class-name)
                                  ; one per argument
                                  ...))))
  (remove-method generic-function method))
```

where *gf-name* is the name of the generic function (i.e. the name of the method), *method-specializers* is either empty or a method combination specifier, such as `:BEFORE`, `:AFTER`, or `:AROUND`, and *parameter-class-name* is the name of the class on which a particular method parameter is specialized.

Package problems; method definitions

Symbol conflicts across packages can be frustrating during development. If you have defined multiple packages for your program, you'll need to be careful to set the proper package (using `IN-PACKAGE`) before defining an object intended for that package. If you inadvertently create an object in the wrong package and then attempt to define it in the correct package, Lisp will signal an error if there is a "uses" relationship between the two packages. The proper response is to first remove the erroneous definition using `UNINTERN`.

You can also get into trouble with packages by having unexported classes defined in two packages and specializing a method based on the wrong class.

The problem with macros

Macros must always be defined before use. This is especially important when you redefine a macro during development: every piece of code that uses the redefined macro `must` be recompiled. You can help yourself avoid macro redefinition problems by reloading your source code after redefining any macro(s).

Runtime tests catch "can't happen cases" when they do...

When I read code, finding the phrase "can't happen" in a comment always raises a red flag. Usually, this statement is made after the programmer has examined the code's execution environment and intended use. Unfortunately, things change and "can't happen" cases do happen.

Lisp provides a very handy facility for checking "can't happen" statements at runtime. The `ASSERT` macro expects a form that will evaluate to true at runtime. If the form evaluates to `NIL` instead, `ASSERT` signals a continuable error, transferring control to the debugger. At the very least, this will help you to learn which assertion was violated so you can correct your program.

`ASSERT` accepts an optional list of value places that the user can interactively change to satisfy the assertion.

```
? (defun add-2 (n)
    (assert (numberp n) (n))
    (+ 2 n))
? (add-2 3)
5
? (add-2 'foo)
Error: Failed assertion (NUMBERP N)
Restart options:
  1. Change the values of some places, then retry the assertion
  2. Top level
  ? 1
  Value for N: 4
6
```

See Chapter 23 for additional information about `ASSERT` and other error detection and recovery techniques.

Use method dispatch rather than case dispatch

When your program needs to make a decision based on the type of an object, you have two choices. You can use `TYPECASE` or `DEFMETHOD`. Unless you have a circumstance that particularly warrants the use of `TYPECASE` (or its variants `CTYPECASE` and `ETYPECASE`) -- and especially if the set of types will change during normal program evolution or maintenance -- you should probably construct your program to operate on the individual types via generic functions. This more clearly exposes the intent of the program and eliminates the likelihood that you will forget to update a `TYPECASE` form during maintenance.

Contents | Cover
Chapter 29 | Chapter 30 | Chapter 31

Copyright © 1995-2001, David B. Lamkins
All Rights Reserved Worldwide

This book may not be reproduced without the written consent of its author. Online distribution is restricted to the author's site.

Chapter 31 - Handling Large Projects in Lisp

This book is primarily a tutorial, designed to give you enough of an understanding of Lisp to get started writing your own programs. Eventually, you'll find yourself in a situation where you need to collaborate with other programmers to implement a larger system. The strategies and tools used to organize Lisp programs for large projects and team efforts are similar to those used for other languages. The main difference is that the coordination tools are part of the same environment in which you develop your program.

Packages keep your names separate from my names

One of the first concerns in group development is to avoid namespace collisions. You don't want to have to poll all the other programmers to make sure that no one has already used the name you're planning to give the routine you're about to write. That would interrupt not only your train of thought, but all the programmers' as well. The alternative -- to ignore the namespace problem and resolve collisions during integration -- is even more unappealing.

One tried and true approach, used in many organizations, is to give every subsystem a unique prefix for its exported names. Your job as a programmer is to tack the proper prefix onto the name of each routine you write for a given subsystem. Like other approaches, this is both annoying and fragile. Prefixes tend to be abbreviations (to save typing); system designers tend to be particularly bad at anticipating future developments -- eventually, you'll have to make exceptions to the prefix naming rule to accommodate new development, and with the exceptions comes the extra mental effort of keeping track of another piece of information which has nothing to do with solving a problem.

Object-based languages at least give you a class scope for naming, but this only pushes the conflict-avoidance strategy somewhere else.

Lisp's package system (see Chapter 3, Lesson 10) lets you partition namespaces independent of other language constructs. If you really want to give each programmer the freedom to create without the overhead of coordinating on matters unrelated to problem-solving, you can give each programmer her own package. As the subsystems are completed, you can integrate by referring to the qualified names of the public APIs of each subsystem. Using this approach, there's no cognitive overhead during subsystem construction, no rework needed during integration, and no runtime overhead in the delivered product.

The keyword package (remember that keywords are symbols with the empty package name, such as `:FOO`) is useful for symbols that are used only for their identity. Without associated code or data, a symbol can readily be shared across all subsystems.

System builders let you describe dependencies

Lisp does not yet have a standard declarative way to describe the process of building a system from its source files. Most projects use one of two approaches:

1. create a system loader based upon `LOAD` (and sometimes, `COMPILE-FILE`) forms
2. use a homegrown or borrowed declarative system builder

Both approaches have their merits. For smaller systems, a naive `LOAD`-based approach is quite workable. As systems get larger, you'll find increasing pressure to reload the minimum set of files necessary to update your working Lisp image from changed sources. And the introduction of macro definitions means that files which use the macros will have to be reloaded whenever the source code for a macro definition changes. Eventually, the complexity of keeping track of these dependencies via ad-hoc loader code will outweigh the pain of constructing, learning, or adapting a declarative system builder.

There are several such programs, collectively referred to as `DEFSYSTEM`s. Some Lisp vendors include a `DEFSYSTEM` with their product. Others are available as source code from Lisp archive sites. Customization or adaptation is usually required for the `DEFSYSTEM`s that are not vendor-supplied; you would be wise to see whether someone has already adapted a `DEFSYSTEM` to your particular environment.

Later in this chapter, we'll see one more way to keep track of file dependencies.

Source control systems keep track of multiple revisions

Did you ever change a file, save it, and then discover that you had broken something so badly that you wanted to go back to the previous version of the file and start over? A source code control system can help you do this.

There is no standard for source code control in Lisp, nor is there likely to be any time soon. Source code control systems are typically provided as an essential programming tool independent of the Lisp environment. Some Lisp vendors offer a way to operate the source code control system from the Lisp environment.

For projects involving more than one programmer, a source code control system offers additional benefits; most such systems allow a programmer to *reserve* a file which she intends to edit. A reserved file can't be edited by any other programmer. Furthermore, the process of reserving a file usually creates a local editable copy for the programmer making the changes; the other programmers see the previous, unedited copy of the file. When the programmer completes (and, of course, tests) the changes, she returns the completed file to the source code control system, which makes the file's new contents available to all the programmers and allows anyone to reserve the file for a new round of updates.

I highly recommend that you take the time to locate and use a source code control system. The effort will pay dividends in the time you don't spend recovering from lost source code changes.

Modules: another way to describe file dependencies

Lisp actually does have a rudimentary system of maintaining file dependencies. I didn't mention the module system earlier because it is deprecated; it might be removed, replaced, or augmented in some future revision of the Lisp specification. I also didn't mention the module system because it has quite limited expressive power. The module system is best suited for finished, stable systems; it does not have enough functionality to support incremental program development in a useful manner. Given all these caveats, let's take a brief look at ...

PROVIDE and REQUIRE

`PROVIDE` and `REQUIRE` are the sole standardized interface to Lisp's module system. A `(REQUIRE` *name*`)` form tells Lisp to see whether the file associated with *name* has already been loaded; if so, `REQUIRE` does nothing, otherwise it loads the file. The loaded file must at some point include a top level form `(PROVIDE` *name*`)`; this informs the module system that the module associated with *name* has been loaded.

The means by which the Lisp system locates the file according to *name* is implementation-dependent; usually the name maps onto a source or object file in the current directory.

The biggest problem with this module system is that it is not designed to handle incremental program changes; it is better suited for loading a completed, stable system. Once a `REQUIRE`d file is loaded, it will never be reloaded. (Your vendor may give you enough information to override this behavior, but you can't depend on it.)

Of course, if you use an ad-hoc loader or a `DEFSYSTEM` during program development, there is little reason to not to deliver the system using the same approach to loading. Better yet, some Lisp environments let you dump an image of your Lisp world, which lets you load the system without having source or object files at all. Either way, there is no good reason to use `PROVIDE` and `REQUIRE`.

Contents | Cover
Chapter 30 | Chapter 31 | Chapter 32

Copyright © 1995-2001, David B. Lamkins
All Rights Reserved Worldwide

This book may not be reproduced without the written consent of its author. Online distribution is restricted to the author's site.

Chapter 32 - Dark Corners and Curiosities

This chapter is almost at the end of our survey of Lisp. Here, we'll examine some Lisp features that are newer, unstandardized, experimental, or controversial.

Extended LOOP: Another little language

Chapter 5 described several iterative control forms: `DO`, `DOTIMES`, `DOLIST`, and a simple `LOOP`. We also saw that `FORMAT` (Chapter 24) has its own control constructs for iteration.

Recursion is a useful tool for describing (and implementing) some algorithms. But in many cases it's easier to write efficient iterative code than it is to write efficient recursive code. In chapters 4 and 28 we saw how to write tail-recursive code, and learned that Lisp is *not* required to optimize tail calls. Ironically, iteration is very important in this implementation of a language originally conceived as a notation for recursive functions.

An *extended* loop facility was introduced late in the specification of Common Lisp. Extended loop, like `FORMAT` control strings, breaks away from the Lisp tradition of a simple, consistent syntax. Extended loop uses keywords to specify initialization, actions and termination conditions. Here are a few examples:

```
;; Sum the integers from 1 to 100
? (loop for n from 1 to 100
        sum n)
5050

;; Compute factorial 10 iteratively
? (loop for n from 1 to 10
        with result = 1
        do (setq result (* result n))
        finally return result)
3628800

;; Gather the even numbers from a list
? (loop for item in '(1 5 8 9 7 2 3)
        when (evenp item)
        collect item)
(8 2)
```

Extended loop inspires heated disagreements among Lisp users. Its detractors point out that the behavior is underspecified for complex combinations of options, while its supporters point out that extended loop forms are easier to read than most `DO` forms for simple operations. You should heed the advice of both camps: use extended loop to improve readability of simple looping operations.

TAGBODY: GO if you must

Ever since the structured programming revolution of the 1970's, programmers and language designers alike have been apologetic about the *GOTO* construct. Yet there are rare cases where a well-placed *GOTO*, used with careful consideration, is the clearest way to structure the control flow of an algorithm.

Lisp retains a *GOTO* as a `GO` form, but it must be used within the lexical scope of a `TAGBODY` form. A `TAGBODY` may contain Lisp forms and symbols. The forms are evaluated, while the symbols (which in other forms might be evaluated for their lexical binding or `SYMBOL-VALUE`) are simply used as labels to which a `GO` form may transfer control.

Processes & Stack Groups: Juggling multiple tasks

Leading-edge Lisp systems on dedicated hardware, and more recently on the Unix platform, have implemented a feature called "lightweight processes." In the C world these are known as "threads."

Lightweight processes allow you to write pieces of code which share the CPU's time along with all of the global variables in your LISP environment. Although this is a limited form of multitasking, lacking protection between processes, it is very useful for handling computations which must run "in the background" or in response to asynchronous events.

In the last few years, low-cost Lisp systems have started to include a process facility. Of all the vendors of low-cost Lisp system, Digitool was the first to include processes in its product. Starting with its 4.0/3.1 release, MCL includes a complete implementation of lightweight processes including a full range of control, synchronization, and communication abstractions. MCL's process API is very close to the API used on the Lisp machines. I'll use MCL's API to illustrate the rest of this section.

The MCL processes are fully preemptive -- you can set both priority and time slice (the "quantum") for each process. Each process can have private variables simply by using local variables in the process run function (i.e., Lisp "closures"). As you'll probably have a need to access shared data as well, the MCL process facility provides locks ("mutexes") to ensure access to critical data by only one process at a time; this is especially useful when multiple fields of a complex structure must be updated in a single operation ("atomically").

The following code implements a solution to Dijkstra's "dining philosophers" problem using MCL processes and locks. In case you're not familiar with this, imagine a group of philosophers seated around a round table. Each philosopher has a plate of food. The food can only be eaten if a philosopher holds a fork in each hand. There is a fork between each pair of philosophers, so there are exactly as many forks as there are philosophers. The objective is to make the philosophers behave so that they all get a fair chance to eat. The classic solution imposes a protocol on how resources (forks) are acquired, in order to prevent deadlock (starvation).

```
(defstruct philosopher
  (amount-eaten 0)
  (task nil))

(defmacro acquire-lock-or-skip (lock post-acquire pre-release &body body)
  `(progn
     ;; Random sleep makes the output more interesting
     ;; by introducing variability into the order of
     ;; execution.  This is a simple way of simulating
     ;; the nondeterminacy that would result from having
     ;; additional processes compete for CPU cycles.
     (sleep (random 5))
     (unless (lock-owner ,lock)
       (process-lock ,lock)
       ,post-acquire
       (unwind-protect
         (progn ,@body)
         ,pre-release
         (process-unlock ,lock)))))

(let ((philosophers #())
      (philosophers-output t))

  (defun dining-philosophers (number-of-philosophers &optional (stream t))
    (unless (equalp philosophers #())
    (stop-philosophers))
  (assert (> number-of-philosophers 1) (number-of-philosophers))
  (setq philosophers-output stream)
  (format philosophers-output
          "~2&Seating ~D philosophers for dinner.~%"
          number-of-philosophers)
  (force-output philosophers-output)
  (flet ((announce-acquire-fork (who fork)
           (format philosophers-output
                   "~&Philosopher ~A has picked up ~A.~%"
                   who (lock-name fork)))
         (announce-release-fork (who fork)
           (format philosophers-output
                   "~&Philosopher ~A is putting down ~A.~%"
                   who (lock-name fork)))
         (eat (who)
           (format philosophers-output
                   "~&Philosopher ~A is EATING bite ~D.~%"
                   who (incf (philosopher-amount-eaten (aref philosophers who))))))
    (flet ((philosopher-task (who left-fork right-fork)
             (loop
               (acquire-lock-or-skip left-fork
                                     (announce-acquire-fork who left-fork)
                                     (announce-release-fork who left-fork)
                 (acquire-lock-or-skip right-fork
                                       (announce-acquire-fork who right-fork)
                                       (announce-release-fork who right-fork)
                   (eat who)))
               (force-output stream)
               (process-allow-schedule))))
      (let ((forks (make-sequence 'vector number-of-philosophers)))
        (dotimes (i number-of-philosophers)
          (setf (aref forks i) (make-lock (format nil "fork ~D" i))))
        (flet ((left-fork (who)
                 (aref forks who))
               (right-fork (who)
                 (aref forks (mod (1+ who) number-of-philosophers))))
          (setq philosophers (make-sequence 'vector number-of-philosophers))
          (dotimes (i number-of-philosophers)
            (setf (aref philosophers i)
                  (make-philosopher
                   :task (process-run-function (format nil "Philosopher-~D" i)
                                               #'philosopher-task
                                               i
                                               (left-fork i)
                                               (right-fork i)))))))))
  (values))

(defun stop-philosophers ()
  (dotimes (i (length philosophers))
    (process-kill (philosopher-task (aref philosophers i))))
  (format philosophers-output
          "~&Dinner is finished. Amounts eaten: {~{~D~^, ~}}~2%"
          (map 'list #'philosopher-amount-eaten philosophers))
  (force-output philosophers-output)
  (setq philosophers #())
  (values))
)
```

If you evaluate `(dining-philosophers 5)` and look through the actions of any one philosopher, you'll see her repeatedly do one of two things:

1. pick up a fork (the left one) and put it down again because the other (right) fork was in use, or
2. pick up each fork (left, then right), eat, then put down the forks.

When you evaluate `(stop-philosophers)` you'll see a list of how many times each philosopher has eaten. These numbers will be fairly close to each other, illustrating the fairness of the algorithm.

MCL also exposes a ``stack group'' abstraction, which is useful for implementing coroutines:

```
;;; Main routine F-FOO
(defun f-foo ()
  (print 'foo-1)
  (funcall sg-bar nil)        ; call 1 to coroutine
  (print 'foo-2)
  (funcall sg-bar nil)        ; call 2 to coroutine
  (print 'foo-3)
  (funcall sg-bar nil)        ; call 3 to coroutine
  nil)

;;; Create a stack group for the coroutine.
(defvar sg-bar (make-stack-group "bar"))

;;; Coroutine F-BAR
(defun f-bar ()
  (print 'bar-1)              ; do this for call 1
  (stack-group-return nil)    ; return from call 1
  (print 'bar-2)              ; do this for call 2
  (stack-group-return nil)    ; return from call 2
  (print 'bar-3)              ; do this for call 3
  (stack-group-return nil)    ; return from call 3
  nil)

;;; Initialization and startup
(defun run-coroutines ()
  ;; Initialize the coroutine
  (stack-group-preset sg-bar #'f-bar)
  ;; Start main coroutine
  (f-foo))
```

When you run the main routine, its execution is interleaved with the coroutine:

```
? (run-coroutines)
FOO-1
BAR-1
FOO-2
BAR-2
FOO-3
BAR-3
NIL
```

You can easily run any function within a separate lightweight process, allowing other computation, compilation, editing, etc. to happen concurrently:

```
(process-run-function "Annoy me"
                      #'(lambda (delay)
                          (loop
                            (sleep delay)
```

```
          (ed-beep)))
   5)
```

Series: Another approach to iteration and filtering

Series were formally introduced with the printing of Common Lisp: The Language (2nd ed) (also known as CLtL2), but were not adopted as part of the ANSI Common Lisp standard. Still, some Lisp vendors include series in their product because customers came to depend upon it during the time between the publication of CLtL2 and the ANSI Specification.

Series combine the behaviors of sequences, streams and loops. Using series, you can write iterative code using a functional notation. Control is achieved by selecting or filtering elements as they pass through a *series* of filters and operators.

The best place to find information and examples is in Appendix A of CLtL2.

Contents | Cover
Chapter 31 | Chapter 32 | Chapter 33

Copyright © 1995-2001, David B. Lamkins
All Rights Reserved Worldwide

This book may not be reproduced without the written consent of its author. Online distribution is restricted to the author's site.

Chapter 33 - Where to Go Next

I hope that this book has whetted your appetite for Lisp. If so, you'll want to explore further; this chapter provides pointers to other sources of information and products.

Suggestions for further reading

The Art of the Metaobject Protocol, Kiczales et al, MIT Press, 1991, ISBN 0-262-61074-4

> This is the definitive text on the metaobject protocol, referred to in Lisp circles as "AMOP." This is *not* light reading; save it for when you feel quite confident in your Lisp abilities.

ANSI Common Lisp, Graham, 1996, Prentice-Hall, ISBN 0-13-370875-6

> This is a good refresher for an experienced Lisp programmer, as well as being an excellent second text for the beginner. (I think it's a bit too terse to use as a first text for a beginner, but you may want to look at it and see whether you think it's approachable.)

On Lisp: Advanced Techniques for Common Lisp, Graham, Prentice Hall, 1994, ISBN 0-13-030552-9

> This has become the canonical reference for macro techniques.

Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS, Keene, 1989, Addison-Wesley, ISBN 0-201-17589-4

> Keene's book is quite simply *the* book to read when you want to understand CLOS. It's short, and it covers all of the essentials. Its best feature is its profuse application of real-world examples.

Understanding CLOS: The Common Lisp Object System, Lawless & Miller, 1991, Digital Press, ISBN 1-55558-064-5

> Lawless and Miller's book covers more of CLOS than Keene's book, but the treatment is closer to a reference than a tutorial.

Common Lisp: The Language, 2nd Ed., Steele, 1990, Digital Press, ISBN 1-55558-041-6

> Dubbed "CLtL2," this was an interim interpretation of the work of the ANSI standardization committee. It has no standing as part of the standards process, but was used by many Lisp vendors to anticipate the final outcome of the committee's work. Some Lisp systems still implement portions of both the CLtL2 description and the ANSI standard.

Garbage Collection: Algorithms for Automatic Dynamic Memory Management, Jones et al, 1996, Wiley, ISBN 0-471-94184-4

> This is an excellent reference covering all aspects of dynamic storage allocation techniques.

Object-Oriented Common Lisp, Slade, 1998, Prentice-Hall, ISBN 0-13-605940-6

> Slade's book is probably the best book available on actually *using* a Common Lisp environment for something other than AI programming. I think it's suitable for a beginner, but should probably be supplemented by another title that provides better insight into the Lisp language.

Common LISPcraft, Wilensky, 1986, W.W. Norton & Co., ISBN 0-393-95544-3

> When I was first learning Common Lisp, I found Wilensky's book the most helpful at exposing some of Lisp's unique concepts, such as closures. This book is easy to read (without being patronizing) and includes a lot of very clear examples. There's also a brief Common Lisp reference in the appendix. I still recommend this as a first book for beginners.

Historical References

Cooperating Sequential Processes, Dijkstra, pp. 43-112 in Programming Languages, Genuys (ed.), Academic Press, 1968.

> Dijkstra described the techniques used for process coordination. The dining philosophers problem is one of Dijkstra's examples of process coordination when resources must be shared.

Recursive Functions of Symbolic Expressions, J. McCarthy, CACM, 3, 4, 1960, ppg. 184-195.

This is McCarthy's seminal Lisp paper. (Available online in various formats at his web site: http://www-formal.stanford.edu/jmc/index.html)

On-line sources

The Association of Lisp Users
Information on conferences, jobs, implementations, etc.
Common Lisp HyperSpec (TM)
A hypertext version of the ANSI Common Lisp standard , constructed by Kent Pittman and hosted by XANALYS (formerly Harlequin, Inc.).
MetaObject Protocol
Not a part of the Common Lisp standard, the MetaObject Protocol is widely supported as a interface to the mechanism underlying CLOS.

Commercial vendors

Digitool, Inc. Home Page
Digitool provides both 68K and PPC versions of Macintosh Common Lisp, a world-class Lisp development system.
The Franz Inc. Home Page
Franz makes Common Lisp compilers for Unix and Windows. They have trial versions for Windows and Linux.
XANALYS
XANALYS (formerly Harlequin) ofers a free Lisp interpreter based upon their LispWorks environment.

Contents | Cover
Chapter 32 | Chapter 33 | Chapter 34

Copyright © 1995-2001, David B. Lamkins
All Rights Reserved Worldwide

This book may not be reproduced without the written consent of its author. Online distribution is restricted to the author's site.

Chapter 34 - Lisp History, or: Origins of Misunderstandings

I'd like to conclude this book with a short history of Lisp's development, providing insights to some lingering misconceptions about Lisp.

John McCarthy's Notation

In the late 1950s, John McCarthy had proposed a mathematical notation to describe recursive functions of symbolic expressions. At this point, McCarthy clearly envisioned a notation rather than a programming language.

Earliest implementations

A couple years later Steve Russell, one of McCarthy's students at MIT, noticed that McCarthy's notation would be easy to interpret on a computer, and LISP was born. While working on an international standards committee to define Algol-60, McCarthy proposed numerous innovations which were not adopted. Perhaps these rejections prompted McCarthy to gather his ideas into what would become Lisp. By late 1959, the first complete Lisp interpreter was in use at MIT.

Special hardware

In the early days, Lisp consistently outstripped the resources of its host machines -- mainframes and minicomputers having memory spaces measured in tens of kilobytes and instruction cycle times measured in microseconds. This prompted Lisp researchers to develop specialized hardware for Lisp program execution. These machines, called "Lisp Machines" (or LispMs) spawned an entire industry that rose in the 1970s and fell during the 1980s. This industry, and not the personal computer industry, was the first to sell personal interactive computers having windowed user interfaces.

Diverging dialects

Concurrent with the rise of the LispM industry, many researchers -- put off by the high costs of the specialized hardware -- engaged in the development of Lisp implementations for stock hardware. This was a time of great diversity and innovation. However, the profusion of dialects prevented researchers from readily sharing their work.

The DARPA directive

DARPA, the Defense Advanced Research Projects Agency, was (and still is) the funding source for much of the Lisp and AI research community. Seeing the problems caused by the explosion in the number of distinct Lisp dialects, DARPA sponsored a project to develop a unified Common Lisp specification.

East vs. West, and European competition

Despite the large number of competing dialects at this time, two were clearly dominant. On the West coast, Interlisp became the standard, with its emphasis on programming aids and tools, such as the aptly named "Do What I Mean." On the East coast, MACLISP was de rigueur, with its focus on low-level system programming access and an efficient compiler.

The Common Lisp effort raised animosities between the two camps, causing most of the Interlisp advocates to withdraw. Also, political forces in Europe prompted the formation of additional standardization efforts, leading to the development of at least one competing (although quite unsuccessfully) standard.

The emergence of compilers for stock hardware

As work began in earnest on the Common Lisp standard, vendors -- most of whom had employees on the standardization committee -- were quick to implement the recommendations under discussion. One of the biggest benefits was the definition of the interface to and behavior of the Lisp compiler; this, together with advances in compiler and garbage collector technology, was a first step toward making Lisp competitive in the arena of general-purpose programming languages.

The long road to standardization

The committee produced the first public edition of the Common LISP specification in 1984. In a shining example of computer mediated cooperative work, hundreds of LISP users and implementers exchanged thousands of email messages to propose, debate, and vote upon each feature of the new language. Each topic and issue was carefully categorized, indexed, and cross-referenced. Very few areas were ambiguous or inadequately specified. Because of the extensive electronic record of the committee's discussions, these remaining areas were clearly identified and served as a basis for continuing work by the committee. An interim report of the committee's work was published in late 1990, and a draft proposed ANSI standard was published in 1992. The X3.226 ANSI Common Lisp standard was finalized in December 1994, and formally published about a year later.

State of the Art?

Lisp has been around in various forms for over forty years. Fortunately, many improvements have been made during that time. Unfortunately, quite a few people in education and industry still think of Lisp as it was twenty or more years in the past.

Today, commercial Lisp implementations have compilers that compete successfully against compilers for lower-level languages such as C and C++. At the same time, C++ has failed to increase its expressive power in a way that competes successfully with Common Lisp.

Java, a newcomer, makes some interesting (yet unfulfilled) promises about portability and security. Java has strengths in the areas of system integration, but struggles with performance and reliability. I believe that Java will carve out a niche for itself only by sacrificing some of its stated goals -- which ones remain to be seen.

Garbage collection, long a favorite whipping post for Lisp detractors, has advanced to the point where collection delays are virtually unnoticeable in a well-written Lisp program. The increasing trend toward server-based applications actually favors Lisp, since garbage collection is more efficient and reliable than manual storage allocation (as with malloc/free) when run for extended periods.

Lisp is still weak on standardized user interfaces, but then so is every other language and platform. Lisp vendors now sell CORBA packages for interoperability and a portable (across Lisp implementations) user interface environment (CLIM). Furthermore, it is easy to write a simple web server in Lisp, allowing the development of user interfaces that run in standard web browsers. A full-featured web server (CL-HTTP) is under continual development at MIT -- the source code is portable to most Lisp platforms and is freely available.

Contents | Cover
Chapter 33 | Chapter 34 | Appendix A

Copyright © 1995-2001, David B. Lamkins
All Rights Reserved Worldwide

This book may not be reproduced without the written consent of its author. Online distribution is restricted to the author's site.

Table of Contents

- About the Author
- About the Book
- Dedication
- Credits
- Copyright
- Acknowledgments
- Foreword
- Introduction

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34
Appendix A

- Chapter 1 - Why Bother? Or: Objections Answered

  Chapter objective: Describe the most common objections to Lisp, and answer each with advice on state-of-the-art implementations and previews of what this book will explain.

  - I looked at Lisp before, and didn't understand it.
  - I can't see the program for the parentheses.
  - Lisp is very slow compared to my favorite language.
  - No one else writes programs in Lisp.
  - Lisp doesn't let me use graphical interfaces.
  - I can't call other people's code from Lisp.
  - Lisp's garbage collector causes unpredictable pauses when my program runs.
  - Lisp is a huge language.
  - Lisp is only for artificial intelligence research.
  - Lisp doesn't have any good programming tools.
  - Lisp uses too much memory.
  - Lisp uses too much disk space.
  - I can't find a good Lisp compiler.

- Chapter 2 - Is this Book for Me?

  Chapter objective: Describe how this book will benefit the reader, with specific examples and references to chapter contents.

  - The Professional Programmer
  - The Student
  - The Hobbyist
  - The Former Lisp Acquaintance
  - The Curious

- Chapter 3 - Essential Lisp in Twelve Lessons

  Chapter objective: Explain Lisp in its simplest form, without worrying about the special cases that can confuse beginners.

Table of Contents

- Lesson 1 - Essential Syntax
    - ◊ Lists are surrounded by parentheses
    - ◊ Atoms are separated by whitespace or parentheses
- Lesson 2 - Essential Evaluation
    - ◊ A form is meant to be evaluated
    - ◊ A function is applied to its arguments
    - ◊ A function can return any number of values
    - ◊ Arguments are usually not modified by a function
    - ◊ Arguments are usually evaluated before function application
    - ◊ Arguments are evaluated in left-to-right order
    - ◊ Special forms and macros change argument evaluation
- Lesson 3 - Some Examples of Special Forms and Macros
    - ◊ SETQ
    - ◊ LET
    - ◊ COND
    - ◊ QUOTE
- Lesson 4 - Putting things together, and taking them apart
    - ◊ CONS
    - ◊ LIST
    - ◊ FIRST and REST
- Lesson 5 - Naming and Identity
    - ◊ A symbol is just a name
    - ◊ A symbol is always unique
    - ◊ A symbol can name a value
    - ◊ A value can have more than one name
- Lesson 6 - Binding versus Assignment
    - ◊ Binding creates a new place to hold a value
    - ◊ Bindings have names
    - ◊ A binding can have different values at the same time
    - ◊ One binding is innermost
    - ◊ The program can only access bindings it creates
    - ◊ Assignment gives an old place a new value
- Lesson 7 - Essential Function Definition
    - ◊ DEFUN defines named functions
    - ◊ LAMBDA defines anonymous functions
- Lesson 8 - Essential Macro Definition
    - ◊ DEFMACRO defines named macros
    - ◊ Macros return a form, not values
- Lesson 9 - Essential Multiple Values
    - ◊ Most forms create only one value
    - ◊ VALUES creates multiple (or no) values
    - ◊ A few special forms receive multiple values
    - ◊ Some forms pass along multiple values
- Lesson 10 - A Preview of Other Data Types
    - ◊ Lisp almost always does the right thing with numbers
    - ◊ Characters give Lisp something to read and write
    - ◊ Arrays organize data into tables
    - ◊ Vectors are one-dimensional arrays
    - ◊ Strings are vectors that contain only characters
    - ◊ Symbols are unique, but they have many values
    - ◊ Structures let you store related data

        - Type information is apparent at runtime
        - Hash Tables provide quick data access from a lookup key
        - Packages keep names from colliding
    - Lesson 11 - Essential Input and Output
        - READ accepts Lisp data
        - PRINT writes Lisp data for you and for READ
        - OPEN and CLOSE let you work with files
        - Variations on a PRINT theme
    - Lesson 12 - Essential Reader Macros
        - The reader turns characters into data
        - Standard reader macros handle built-in data types
        - User programs can define reader macros

- Chapter 4 - Mastering the Essentials

Chapter objective: Reinforce the concepts of Chapter 3 with hands-on exercises.

    - Hands-on! The "toploop"
    - Spotting and avoiding common mistakes
    - Defining simple functions
    - Using global variables and constants
    - Defining recursive functions
    - Tail recursion
    - Exercises in naming
    - Lexical binding and multiple name spaces
    - Reading, writing, and arithmetic
    - Other data types
    - Simple macros
    - Reader macros

- Chapter 5 - Introducing Iteration

Chapter objective: Introduce the most common looping constructs.

> "There's no such thing as an infinite loop. Eventually, the computer will break." --
> John D. Sullivan

    - Simple LOOP loops forever...
    - But there's a way out!
    - Use DOTIMES for a counted loop
    - Use DOLIST to process elements of a list
    - DO is tricky, but powerful

- Chapter 6 - Deeper into Structures

Chapter objective: Show the most useful optional features of structures.

    - Default values let you omit some initializers, sometimes
    - Change the way Lisp prints your structures
    - Alter the way structures are stored in memory

  - Shorten slot accessor names
  - Allocate new structures without using keyword arguments
  - Define one structure as an extension of another

- Chapter 7 - A First Look at Objects as Fancy Structures

  Chapter objective: Introduce CLOS objects as tools for structuring data. Object behaviors will be covered in a later chapter.

  - Hierarchies: Classification vs. containment
  - Use DEFCLASS to define new objects
  - Objects have slots, with more options than structures
  - Controlling access to a slot helps keep clients honest
  - Override a slot accessor to do things that the client can't
  - Define classes with single inheritance for specialization
  - Multiple inheritance allows mix-and-match definition
  - Options control initialization and provide documentation
  - This is only the beginning...

- Chapter 8 - Lifetime and Visibility

  Chapter objective: Show how lifetime and visibility affect the values of Lisp variables during execution. This is pretty much like local and global variables in other languages, but Lisp's special variables change things. This chapter also sets the stage for understanding that lifetime and visibility isn't just for variables.

  - Everything in Lisp has both lifetime and visibility
  - Lifetime: Creation, existence, then destruction
  - Visibility: To see and to be seen by
  - The technical names: Extent and Scope
  - Really easy cases: top-level defining forms
  - Scope and extent of parameters and LET variables
  - Slightly trickier: special variables

- Chapter 9 - Introducing Error Handling and Non-Local Exits

  Chapter objective: Show three new ways of transferring control between parts of a program.

  - UNWIND-PROTECT: When it absolutely, positively has to run
  - Gracious exits with BLOCK and RETURN-FROM
  - Escape from anywhere (but not at any time) with CATCH and THROW
  - Making sure files only stay open as long as needed

- Chapter 10 - How to Find Your Way Around, Part 1

  Chapter objective: Show how to find things in Lisp without resorting to the manual.

  - APROPOS: I don't remember the name, but I recognize the face
  - DESCRIBE: Tell me more about yourself
  - INSPECT: Open wide and say "Ah..."

  - DOCUMENTATION: I know I wrote that down somewhere

- Chapter 11 - Destructive Modification

  Chapter objective: Illustrate the difference between assignment and binding, and show why assignment is harder to understand. Then explore reasons for preferring the more difficult concept, and introduce functions that use destructive modification.

  - Simple assignment is destructive modification
  - The risk of assignment
  - Changing vs. copying: an issue of efficiency
  - Modifying lists with destructive functions
  - RPLACA, RPLACD, SETF ...; circularity
  - Places vs. values: destructive functions don't always have the desired side-effect
  - Contrast e.g. PUSH and DELETE
  - Shared and constant data: Dangers of destructive changes

- Chapter 12 - Mapping Instead of Iteration

  Chapter objective: Categorize and demonstrate the mapping functions. Show appropriate and inappropriate uses. Introduce the concept of sequences.

  - MAPCAR, MAPC, and MAPCAN process successive list elements
  - MAPLIST, MAPL, and MAPCON process successive sublists
  - MAP and MAP-INTO work on sequences, not just lists
  - Mapping functions are good for filtering
  - It's better to avoid mapping if you care about efficiency
  - Predicate mapping functions test sequences
  - SOME, EVERY, NOTANY, NOTEVERY
  - REDUCE combines sequence elements

- Chapter 13 - Still More Things You Can Do with Sequences

  Chapter objective: Introduce the most useful sequence functions, and show how to use them. Show how destructive sequence functions must be used to have the intended effect.

  - CONCATENATE: new sequences from old
  - ELT and SUBSEQ get what you want from any sequence (also, COPY-SEQ)
  - REVERSE turns a sequence end-for-end (also, NREVERSE)
  - LENGTH: size counts after all
  - COUNT: when it's what's inside that matters
  - REMOVE, SUBSTITUTE, and other sequence changers
  - DELETE, REMOVE-DUPLICATES, DELETE-DUPLICATES, and NSUBSTITUTE
  - FILL and REPLACE
  - Locating things in sequences: POSITION, FIND, SEARCH, and MISMATCH
  - SORT and MERGE round out the sequence toolkit

- Chapter 14 - Can Objects Really Behave Themselves?

Chapter objective: Show how generic functions work. Describe multiple dispatch, inheritance, and method combination. Preview the metaobject protocol.

  - Generic functions give objects their behaviors
  - The line between methods and objects blurs for multimethods
  - Methods on non-objects? So where does the method live?
  - Generic functions work by dispatching on argument specializers
  - Object inheritance matters after all; finding the applicable method
  - Method combinations offer further choices
  - Nothing is cast in stone; a peek at the metaobject protocol

- Chapter 15 - Closures

Chapter objective: Show how closures capture free variables for use in other execution contexts. Demonstrate with some practical applications.

  - Is it a function of the lifetime, or the lifetime of a function?
  - How to spot a free variable, and what to do about it.
  - Using closures to keep private, secure information.
  - Functions that return functions, and how they differ from macros.

- Chapter 16 - How to Find Your Way Around, Part 2

Chapter objective: Learn what the Lisp compiler does to your code, and how to watch what your code does as it runs.

> "DISASSEMBLE is your friend." -- Bill St. Clair

  - DISASSEMBLE: I always wondered what they put inside those things...
  - BREAK and backtrace: How did I end up here?
  - TRACE and STEP: I'm watching you!

- Chapter 17 - Not All Comparisons are Equal

Chapter objective: Tell how and why Lisp has so many different comparison functions. Give guidelines for proper choice.

  - The longer the test, the more it tells you
  - EQ is true for identical symbols
  - EQL is also true for identical numbers and characters
  - EQUAL is usually true for things that print the same
  - EQUALP ignores number type and character case
  - Longer tests are slower; know what you're comparing
  - Specialized tests run faster on more restricted data types

- Chapter 18 - Very Logical, Indeed...

Chapter objective: Describe common logical functions, and conditional evaluation. Introduce bit manipulation functions, bit vectors, and generalized byte manipulation.

- ♦ AND and OR evaluate only as much as they need
- ♦ Bits, bytes, and Boole
- ♦ Bit vectors can go on forever
- ♦ Chunks of bits make bytes

- Chapter 19 - Streams

Chapter objective: Introduce streams as generalized I/O facilities. Emphasize interchangeability of streams attached to different devices.

- ♦ Streams provide a pipe to supply or accept data
- ♦ Creating streams on files
- ♦ Creating streams on strings
- ♦ Binary I/O

- Chapter 20 - Macro Etiquette

Chapter objective: Go beyond the simple examples of chapters 3 and 4 to show how to properly construct macros to solve a wide variety of problems.

- ♦ Macros are programs that generate programs
- ♦ Close up: how macros work
- ♦ Backquote looks like a substitution template
- ♦ Beyond the obvious, part 1: compute, then generate
- ♦ Beyond the obvious, part 2: macros that define macros
- ♦ Tricks of the trade: elude capture using GENSYM
- ♦ Macros vs. inlining

- Chapter 21 - Fancy Tricks with Function and Macro Arguments

Chapter objective: Describe lambda-list options. Show how they can be used to clarify programs.

- ♦ Keywords let you name your parameters
- ♦ Default values for when you'd rather not say
- ♦ Add some structure to your macros by taking apart arguments

---

- Chapter 22 - How to Find Your Way Around, Part 3

Chapter objective: Learn how to find out about objects and methods. Learn specialized techniques to alter or monitor program behavior without changing the source code.

- ♦ Class and method browsers help you find your way in a sea of objects
- ♦ ADVISE lets you modify a function's behavior without changing the function
- ♦ WATCH lets you open a window on interesting variables

- Chapter 23 - To Err is Expected; To Recover, Divine

Chapter objective: Show how to create your own error detection and recovery mechanisms.

- ♦ Signal your own errors and impress your users

  - Categorize errors using Conditions
  - Recover from Conditions using Restarts

- Chapter 24 - FORMAT Speaks a Different Language

Chapter objective: Describe the most useful functions of the FORMAT function.

  - FORMAT rhymes with FORTRAN, sort of...
  - Formatting
  - Iteration
  - Conditionals
  - Floobydust

- Chapter 25 - Connecting Lisp to the Real World

Chapter objective: Describe FFI in general, and give examples from actual implementations. Show how to use wrappers to call C++ functions. Show how callbacks allow C programs to call Lisp functions. Give an example using TCP/IP access.

  - Foreign Function Interfaces let you talk to programs written in "foreign languages"
  - Would you wrap this, please?
  - I'll call you back...
  - Network Interfaces: beyond these four walls

- Chapter 26 - Put on a Happy Face: Interface Builders

Chapter objective: Discuss event-driven interfaces and GUI builders in general, then show examples from major desktop Common Lisp platforms. Conclude with a discussion of platform-independent Lisp GUIs such as Garnet and CLIM.

  - Event-driven interfaces
  - Graphical programming
  - Example: MCL's Interface Toolkit
  - Platform-independent interfaces

- Chapter 27 - A Good Editor is Worth a Thousand Keystrokes

Chapter objective: Show how Lisp's simple syntax combines with an integrated editor to ease many of the common tasks of writing a Lisp program.

  - Simple syntax; smart editors
  - Matching and flashing
  - Automatic indentation
  - Symbol completion
  - Finding definitions
  - On-line documentation
  - Access to debugging tools
  - Extending the editor using Lisp

- Chapter 28 - Practical Techniques for Programming

Chapter objective: Provide useful guidelines for Lisp style. Give practical advice on tradeoffs among debugging, performance, and readability.

- Elements of Lisp style
- Property lists are handy for small (very small) ad-hoc databases
- Declarations help the compiler, sometimes
- DEFVAR versus DEFPARAMETER
- Define constants with DEFCONSTANT
- Know when (not) to use the compiler
- Speed vs. ability to debug
- Efficiency: spotting it, testing it
- Recognizing inefficiency, profiling; performance vs. readability

- Chapter 29 - I Thought it was Your Turn to Take Out the Garbage

Chapter objective: Describe the benefits and costs of garbage collection. Show how to improve program performance by reducing the amount of garbage it generates.

- What is garbage?
- Why is garbage collection important?
- How does garbage collection work?
- What effect does garbage have on my program?
- How can I reduce garbage collection pauses in my program?

- Chapter 30 - Helpful Hints for Debugging and Bug-Proofing

Chapter objective: Describe use of Lisp's debugging tools.

- Finding the cause of an error
- Reading backtraces, compiler settings for debugging
- Simple debugging tools
- BREAK, PRINT
- Power tools for tough problems
- TRACE, STEP, ADVISE, WATCH
- Into the belly of the beast
- INSPECT, DESCRIBE
- Continuing from an error
- Problems with unwanted definitions
- Package problems; method definitions
- The problem with macros
- Runtime tests catch "can't happen cases" when they do...
- Use method dispatch rather than case dispatch

- Chapter 31 - Handling Large Projects in Lisp

Chapter objective: Describe strategies and tools used to organize Lisp programs for large projects and team efforts.

- Packages keep your names separate from my names
- System builders let you describe dependencies

- Source control systems keep track of multiple revisions
- Modules: another way to describe file dependencies
- PROVIDE and REQUIRE

- Chapter 32 - Dark Corners and Curiosities

Chapter objective: Describe some Lisp features that are newer, unstandardized, experimental, or controversial.

- Extended LOOP: Another little language
- TAGBODY: GO if you must
- Processes & Stack Groups: Juggling multiple tasks
- Series: Another approach to iteration and filtering

- Chapter 33 - Where to Go Next

Chapter objective: Provide pointers to other sources of information and products.

- Suggestions for further reading
- On-line sources
- Commercial vendors

- Chapter 34 - Lisp History, or: Origins of Misunderstandings

Chapter objective: Give a short history of Lisp's development, providing insights to some lingering misconceptions.

- John McCarthy's Notation
- Earliest implementations
- Special hardware
- Diverging dialects
- The DARPA directive
- East vs. West, and European competition
- The emergence of compilers for stock hardware
- The long road to standardization
- State of the Art?

- Appendix A - Successful Lisp Applications

Chapter objective: Describe large successful applications built in Lisp.

- Emacs
- G2
- AutoCad
- Igor Engraver
- Yahoo Store
- ...

Cover
About the Author

Table of Contents

Copyright © 1995-2001, David B. Lamkins
All Rights Reserved Worldwide

This book may not be reproduced without the written consent of its author. Online distribution is restricted to the author's site.

Copyright

This book is copyright © 1995-2001 David B. Lamkins. All rights are reserved worldwide.

Contents | Cover
Credits | Copyright | Acknowledgments

Copyright © 1995-2001, David B. Lamkins
All Rights Reserved Worldwide

This book may not be reproduced without the written consent of its author. Online distribution is restricted to the author's site.

Copyright

Successful Lisp:
How to Understand and Use Common Lisp

David B. Lamkins
david@lamkins.net

- Provides an overview of Common Lisp for the working programmer.
- Introduces key concepts in an easy-to-read format.
- Describes format, typical use, and possible drawbacks of all important Lisp constructs.
- Provides practical advice for the construction of Common Lisp programs.
- Shows examples of how Common Lisp is best used.
- Illustrates and compares features of the most popular Common Lisp systems on desktop computers.
- Includes discussion and examples of advanced constructs for iteration, error handling, object oriented programming, graphical user interfaces, and threading.
- Supplements Common Lisp reference books and manuals with useful hands-on techniques.
- Shows how to find what you need among the thousands of documented and undocumented functions and variables in a typical Common Lisp system.

Frequently asked questions:

1. *May I make a copy of the online version of Successful Lisp for my own use?*

   Yes, you may make a copy this HTML version so long as your copy is offline. You may not publish Successful Lisp on any network accessible to the public, nor may you distribute offline copies in any form.
2. *Is Successful Lisp available in Postscript, PDF, or an archive format such as tar, zip or sit?*

   No. I maintain the content in the web form that you see here. Making an archival version would be contrary to my preference of having readers link directly to my site so as to always have access to the latest version of the text.
3. *Is Successful Lisp available in a printed or printable format?*

   Yes! A print edition is available as of December 2004 (ISBN 3-937526-00-5). The book contains numerous corrections and improved typography and layout. You can order Successful Lisp from Amazon.com, or check with your local bookstore or other major online booksellers.
4. *I'm having difficulty running my Lisp system or writing a Lisp program. Can you help?*

   No. I don't have enough free time to engage in conversations of this nature. I suggest that you talk to your Lisp vendor or solicit the help of the fine people who frequent the `comp.lang.lisp` newsgroup. Be aware that these people take a dim view of students who ask for homework solutions unless the student has made (and documented) an attempt to solve the problem. Students will find that they receive a warmer welcome in `comp.lang.lisp` if they ask a specific question related to understanding a particular aspect of Lisp.

Contents

Copyright © 1995-2001, David B. Lamkins
All Rights Reserved Worldwide

This book may not be reproduced without the written consent of its author. Online distribution is restricted to the author's site.

Successful Lisp: How to Understand and Use Common Lisp

David B. Lamkins
david@lamkins.net

Printed edition now available!

This book:

- Provides an overview of Common Lisp for the working programmer.
- Introduces key concepts in an easy-to-read format.
- Describes format, typical use, and possible drawbacks of all important Lisp constructs.
- Provides practical advice for the construction of Common Lisp programs.
- Shows examples of how Common Lisp is best used.
- Illustrates and compares features of the most popular Common Lisp systems on desktop computers.
- Includes discussion and examples of advanced constructs for iteration, error handling, object oriented programming, graphical user interfaces, and threading.
- Supplements Common Lisp reference books and manuals with useful hands-on techniques.
- Shows how to find what you need among the thousands of documented and undocumented functions and variables in a typical Common Lisp system.

Frequently asked questions:

1. *May I make a copy of the online version of Successful Lisp for my own use?*

   Yes, you may make a copy this HTML version so long as your copy is offline. You may not publish Successful Lisp on any network accessible to the public, nor may you distribute offline copies in any form.
2. *Is Successful Lisp available in Postscript, PDF, or an archive format such as tar, zip or sit?*

   No. I maintain the content in the web form that you see here. Making an archival version would be contrary to my preference of having readers link directly to my site so as to always have access to the latest version of the text.
3. *Is Successful Lisp available in a printed or printable format?*

   Yes! A print edition is available as of December 2004 (ISBN 3-937526-00-5). The book contains numerous corrections and improved typography and layout. You can order Successful Lisp from Amazon.com, or check with your local bookstore or other major online booksellers.
4. *I'm having difficulty running my Lisp system or writing a Lisp program. Can you help?*

   No. I don't have enough free time to engage in conversations of this nature. I suggest that you talk to your Lisp vendor or solicit the help of the fine people who frequent the `comp.lang.lisp` newsgroup. Be aware that these people take a dim view of students who ask for homework solutions unless the student has made (and documented) an attempt to solve the problem. Students will find that they receive a warmer welcome in `comp.lang.lisp` if they ask a specific question related to understanding a particular aspect of Lisp.

Contents

Copyright © 1995-2001, David B. Lamkins
All Rights Reserved Worldwide

This book may not be reproduced without the written consent of its author. Online distribution is restricted to the author's site.

Credits

Trademarks mentioned in this book are the property of their respective owners.

Ray Scanlon generously donated his proofreading services for the entire book. All errors that remain are probably due to my ignoring Ray's advice or have been introduced since his reading.

Mary-Suzanne donated the good-looking graphics used in the book. The graphics that look amateurish are mine. Never let an engineer attempt to do graphic arts.

Contents | Cover
Dedication | Credits | Copyright

Copyright © 1995-2001, David B. Lamkins
All Rights Reserved Worldwide

This book may not be reproduced without the written consent of its author. Online distribution is restricted to the author's site.

Dedication

This book is dedicated to my sons Nick and Ian. I'm very proud of you guys. Follow your dreams!

Contents | Cover
About the Book | Dedication | Credits

Copyright © 1995-2001, David B. Lamkins
All Rights Reserved Worldwide

This book may not be reproduced without the written consent of its author. Online distribution is restricted to the author's site.

Foreword

I started writing this book six years ago in response to a publisher's inquiry about Lisp books. Part of their submission process involved my filling out what amounted to a market research form that disclosed all of the Lisp books I knew about, their publication dates, and a brief synopsis of the strengths and weaknesses of each.

On the basis of my market research, the publisher decided that their marketplace didn't need another Lisp book. So I kept going, because I knew otherwise. I wrote in fits and starts over the first two years, and published an early draft of the book on the web. Readers of "Successful Lisp" from all over the world have sent me positive feedback, thanking me for making the book available as a resource for their use in classes and personal studies of Common Lisp.

A few of the more enthusiastic readers even compared "Successful Lisp" to a couple of my favorite Lisp texts. While I'll admit to having my spirits buoyed by such unabashed enthusiam, I'll also be the first to point out that "Successful Lisp" attempts to cover the subject in a somewhat different manner, and at different levels of detail, than the other available texts. By all means, enjoy this book. But when you need more information than I've been able to fit in this limited space, please turn to some of the other fine books listed in Chapter 33.

Common Lisp is, at its core, a very simple language. Its apparent size is due not to complexity (as is the case with certain more recent languages) but rather to the breadth of functionality implemented via the functions and data types that are provided in every copy of Common Lisp.

The other encouraging feature of Common Lisp is its stability. The language became an ANSI standard in 1994 after four years of intensive work by vendors and designers alike, during which time several subtle problems and inconsistencies were removed from the language and the corrections implemented in production compilers and tested against real-world applications. This time consuming process of review and refinement was quite successful, in that the language has not required correction, change or clarification since its standardization. That's good news for me, since I haven't had to revise my book to keep up. Good news, too, for the people who write large, complex programs in Common Lisp; their code just keeps on working even when they change hardware or compilers.

The one criticism that has arisen over the years is that Common Lisp hasn't adopted enough cool new functionality to give it more of a mass appeal. Vendors provide their own extensions for networking, graphics, multiprocessing and other features, but the lack of standardization makes it difficult to employ these features in a portable manner. While I share some of that concern, I'll also observe that these very features have changed significantly over the years since the ANSI standardization of Common Lisp. Over the same period, newer languages have borrowed from Common Lisp's toolbox for ideas regarding expression of algorithms, symbolic manipulation of data, and automatic storage management. Someday networking and graphics will be as well defined, and I'm sure we'll see these aspects of computing incorporated into Common Lisp. For now, be glad that you can tell the difference between what's stable and what's the flavor of the month.

This will probably be my last update to "Successful Lisp", as my personal goals and interests have taken me away from the deep involvement I had with computing through the 80s and 90s. I suspect that "Successful Lisp", if it has a lasting value within the Common Lisp community, will do so because of the stability and longevity of the language itself.

I wish you well. Enjoy the book.

David Lamkins
February 2001

Contents | Cover
Acknowledgments | Foreword | Introduction

Copyright © 1995-2001, David B. Lamkins
All Rights Reserved Worldwide

This book may not be reproduced without the written consent of its author. Online distribution is restricted to the author's site.

Introduction

Lisp has a long, rich history dating back more than forty years. It has survived all of the programming "revolutions" that have rendered lesser langauges obsolete. Despite its being taught as a curiosity, if at all, by college and university staff who themselves have a poor understanding of the continuing growth and evolution of Lisp, new generations of programmers continue to seek out Lisp as a tool to solve some of the most difficult problems in the world of computing.

This book is my attempt to help the current generation of Lisp programmers, and to give something back to those who have paved the way for the rest of us.

The first two chapters lay out the book's background and intent. Then chapters 3 through 24 introduce a core subset of Common Lisp. Chapters 25 through 32 introduce useful features that may not be found in all Lisp implementations, or their details may differ among implementations; these chapters should be read in conjunction with your Lisp vendor's documentation. Chapters 33 and 34 offer, respectively, an annotated list of additional resources and an overview of Lisp's historical timeline.

Contents | Cover
Foreword | Introduction | Chapter 1

Copyright © 1995-2001, David B. Lamkins
All Rights Reserved Worldwide

This book may not be reproduced without the written consent of its author. Online distribution is restricted to the author's site.

Professional Track Suggested Chapter List

Skim for basics:

- Chapter 3 - Essential Lisp in Twelve Lessons
- Chapter 4 - Mastering the Essentials
- Chapter 5 - Introducing Iteration
- Chapter 6 - Deeper into Structures

Read for new concepts:

- Chapter 7 - A First Look at Objects as Fancy Structures
- Chapter 8 - Lifetime and Visibility
- Chapter 9 - Introducing Error Handling and Non-Local Exits
- Chapter 11 - Destructive Modification
- Chapter 12 - Mapping Instead of Iteration
- Chapter 13 - Still More Things You Can Do with Sequences
- Chapter 14 - Can Objects Really Behave Themselves?
- Chapter 15 - Closures
- Chapter 17 - Not All Comparisons are Equal
- Chapter 18 - Very Logical, Indeed
- Chapter 19 - Streams
- Chapter 20 - Macro Etiquette
- Chapter 21 - Fancy Tricks with Function and Macro Arguments
- Chapter 23 - To Err is Expected; To Recover, Divine
- Chapter 24 - FORMAT Speaks a Different Language

Learn the environment and tools:

- Chapter 10 - How to Find Your Way Around, Part 1
- Chapter 16 - How to Find Your Way Around, Part 2
- Chapter 22 - How to Find Your Way Around, Part 3
- Chapter 26 - Put on a Happy Face: Interface Builders
- Chapter 27 - A Good Editor is Worth a Thousand Keystrokes
- Chapter 28 - Practical Techniques for Programming

Additional topics:

- Chapter 25 - Connecting Lisp to the Real World
- Chapter 29 - I Thought it was Your Turn to Take Out the Garbage
- Chapter 30 - Helpful Hints for Debugging and Bug-Proofing
- Chapter 31 - Handling Large Projects in Lisp
- Chapter 32 - Dark Corners and Curiosities
- Chapter 33 - Where to Go Next
- Chapter 34 - Lisp History, or: Origins of Misunderstandings
- Appendix A - Successful Lisp Applications

Contents | Cover
Chapter 2

Copyright © 1995-2001, David B. Lamkins
All Rights Reserved Worldwide

This book may not be reproduced without the written consent of its author. Online distribution is restricted to the author's site.

Student Track Suggested Chapter List

Learn the whole language:

- Chapter 3 - Essential Lisp in Twelve Lessons
- Chapter 4 - Mastering the Essentials
- Chapter 5 - Introducing Iteration
- Chapter 6 - Deeper into Structures
- Chapter 7 - A First Look at Objects as Fancy Structures
- Chapter 8 - Lifetime and Visibility
- Chapter 9 - Introducing Error Handling and Non-Local Exits
- Chapter 10 - How to Find Your Way Around, Part 1
- Chapter 11 - Destructive Modification
- Chapter 12 - Mapping Instead of Iteration
- Chapter 13 - Still More Things You Can Do with Sequences
- Chapter 14 - Can Objects Really Behave Themselves?
- Chapter 15 - Closures
- Chapter 16 - How to Find Your Way Around, Part 2
- Chapter 17 - Not All Comparisons are Equal
- Chapter 18 - Very Logical, Indeed
- Chapter 19 - Streams
- Chapter 20 - Macro Etiquette
- Chapter 21 - Fancy Tricks with Function and Macro Arguments
- Chapter 22 - How to Find Your Way Around, Part 3
- Chapter 23 - To Err is Expected; To Recover, Divine
- Chapter 24 - FORMAT Speaks a Different Language

Read as interested:

- Chapter 25 - Connecting Lisp to the Real World
- Chapter 26 - Put on a Happy Face: Interface Builders
- Chapter 27 - A Good Editor is Worth a Thousand Keystrokes
- Chapter 28 - Practical Techniques for Programming
- Chapter 29 - I Thought it was Your Turn to Take Out the Garbage
- Chapter 30 - Helpful Hints for Debugging and Bug-Proofing
- Chapter 31 - Handling Large Projects in Lisp
- Chapter 32 - Dark Corners and Curiosities
- Chapter 33 - Where to Go Next
- Chapter 34 - Lisp History, or: Origins of Misunderstandings
- Appendix A - Successful Lisp Applications

Contents | Cover
Chapter 2

Copyright © 1995-2001, David B. Lamkins
All Rights Reserved Worldwide

This book may not be reproduced without the written consent of its author. Online distribution is restricted to the author's site.

Hobbyist Track Suggested Chapter List

Learn how to find your way:

- Chapter 10 - How to Find Your Way Around, Part 1
- Chapter 16 - How to Find Your Way Around, Part 2
- Chapter 22 - How to Find Your Way Around, Part 3
- Chapter 28 - Practical Techniques for Programming
- Chapter 29 - I Thought it was Your Turn to Take Out the Garbage
- Chapter 30 - Helpful Hints for Debugging and Bug-Proofing

Read as interested:

- Chapter 3 - Essential Lisp in Twelve Lessons
- Chapter 4 - Mastering the Essentials
- Chapter 5 - Introducing Iteration
- Chapter 6 - Deeper into Structures
- Chapter 7 - A First Look at Objects as Fancy Structures
- Chapter 8 - Lifetime and Visibility
- Chapter 9 - Introducing Error Handling and Non-Local Exits
- Chapter 11 - Destructive Modification
- Chapter 12 - Mapping Instead of Iteration
- Chapter 13 - Still More Things You Can Do with Sequences
- Chapter 14 - Can Objects Really Behave Themselves?
- Chapter 15 - Closures
- Chapter 17 - Not All Comparisons are Equal
- Chapter 18 - Very Logical, Indeed
- Chapter 19 - Streams
- Chapter 20 - Macro Etiquette
- Chapter 21 - Fancy Tricks with Function and Macro Arguments
- Chapter 23 - To Err is Expected; To Recover, Divine
- Chapter 24 - FORMAT Speaks a Different Language
- Chapter 25 - Connecting Lisp to the Real World
- Chapter 26 - Put on a Happy Face: Interface Builders
- Chapter 27 - A Good Editor is Worth a Thousand Keystrokes
- Chapter 31 - Handling Large Projects in Lisp
- Chapter 32 - Dark Corners and Curiosities
- Chapter 33 - Where to Go Next
- Chapter 34 - Lisp History, or: Origins of Misunderstandings
- Appendix A - Successful Lisp Applications

Contents | Cover
Chapter 2

Copyright © 1995-2001, David B. Lamkins
All Rights Reserved Worldwide

This book may not be reproduced without the written consent of its author. Online distribution is restricted to the author's site.

Former User Track Suggested Chapter List

Read as needed:

- Chapter 3 - Essential Lisp in Twelve Lessons
- Chapter 4 - Mastering the Essentials
- Chapter 5 - Introducing Iteration
- Chapter 6 - Deeper into Structures
- Chapter 7 - A First Look at Objects as Fancy Structures
- Chapter 8 - Lifetime and Visibility
- Chapter 9 - Introducing Error Handling and Non-Local Exits
- Chapter 10 - How to Find Your Way Around, Part 1
- Chapter 11 - Destructive Modification
- Chapter 12 - Mapping Instead of Iteration
- Chapter 13 - Still More Things You Can Do with Sequences
- Chapter 14 - Can Objects Really Behave Themselves?
- Chapter 15 - Closures
- Chapter 16 - How to Find Your Way Around, Part 2
- Chapter 17 - Not All Comparisons are Equal
- Chapter 18 - Very Logical, Indeed
- Chapter 19 - Streams
- Chapter 20 - Macro Etiquette
- Chapter 21 - Fancy Tricks with Function and Macro Arguments
- Chapter 22 - How to Find Your Way Around, Part 3
- Chapter 23 - To Err is Expected; To Recover, Divine
- Chapter 24 - FORMAT Speaks a Different Language
- Chapter 25 - Connecting Lisp to the Real World
- Chapter 26 - Put on a Happy Face: Interface Builders
- Chapter 27 - A Good Editor is Worth a Thousand Keystrokes
- Chapter 28 - Practical Techniques for Programming
- Chapter 29 - I Thought it was Your Turn to Take Out the Garbage
- Chapter 30 - Helpful Hints for Debugging and Bug-Proofing
- Chapter 31 - Handling Large Projects in Lisp
- Chapter 32 - Dark Corners and Curiosities
- Chapter 33 - Where to Go Next
- Chapter 34 - Lisp History, or: Origins of Misunderstandings
- Appendix A - Successful Lisp Applications

Also see detailed Table of Contents.

Contents | Cover
Chapter 2

Copyright © 1995-2001, David B. Lamkins
All Rights Reserved Worldwide

This book may not be reproduced without the written consent of its author. Online distribution is restricted to the author's site.

Curious Reader Track Suggested Chapter List

Read in order, skimming as desired:

- Chapter 3 - Essential Lisp in Twelve Lessons
- Chapter 4 - Mastering the Essentials
- Chapter 5 - Introducing Iteration
- Chapter 6 - Deeper into Structures
- Chapter 7 - A First Look at Objects as Fancy Structures
- Chapter 8 - Lifetime and Visibility
- Chapter 9 - Introducing Error Handling and Non-Local Exits
- Chapter 10 - How to Find Your Way Around, Part 1
- Chapter 11 - Destructive Modification
- Chapter 12 - Mapping Instead of Iteration
- Chapter 13 - Still More Things You Can Do with Sequences
- Chapter 14 - Can Objects Really Behave Themselves?
- Chapter 15 - Closures
- Chapter 16 - How to Find Your Way Around, Part 2
- Chapter 17 - Not All Comparisons are Equal
- Chapter 18 - Very Logical, Indeed
- Chapter 19 - Streams
- Chapter 20 - Macro Etiquette
- Chapter 21 - Fancy Tricks with Function and Macro Arguments
- Chapter 22 - How to Find Your Way Around, Part 3
- Chapter 23 - To Err is Expected; To Recover, Divine
- Chapter 24 - FORMAT Speaks a Different Language
- Chapter 25 - Connecting Lisp to the Real World
- Chapter 26 - Put on a Happy Face: Interface Builders
- Chapter 27 - A Good Editor is Worth a Thousand Keystrokes
- Chapter 28 - Practical Techniques for Programming
- Chapter 29 - I Thought it was Your Turn to Take Out the Garbage
- Chapter 30 - Helpful Hints for Debugging and Bug-Proofing
- Chapter 31 - Handling Large Projects in Lisp
- Chapter 32 - Dark Corners and Curiosities
- Chapter 33 - Where to Go Next
- Chapter 34 - Lisp History, or: Origins of Misunderstandings
- Appendix A - Successful Lisp Applications

Contents | Cover
Chapter 2

Copyright © 1995-2001, David B. Lamkins
All Rights Reserved Worldwide

This book may not be reproduced without the written consent of its author. Online distribution is restricted to the author's site.

Curious Reader Track Suggested Chapter List