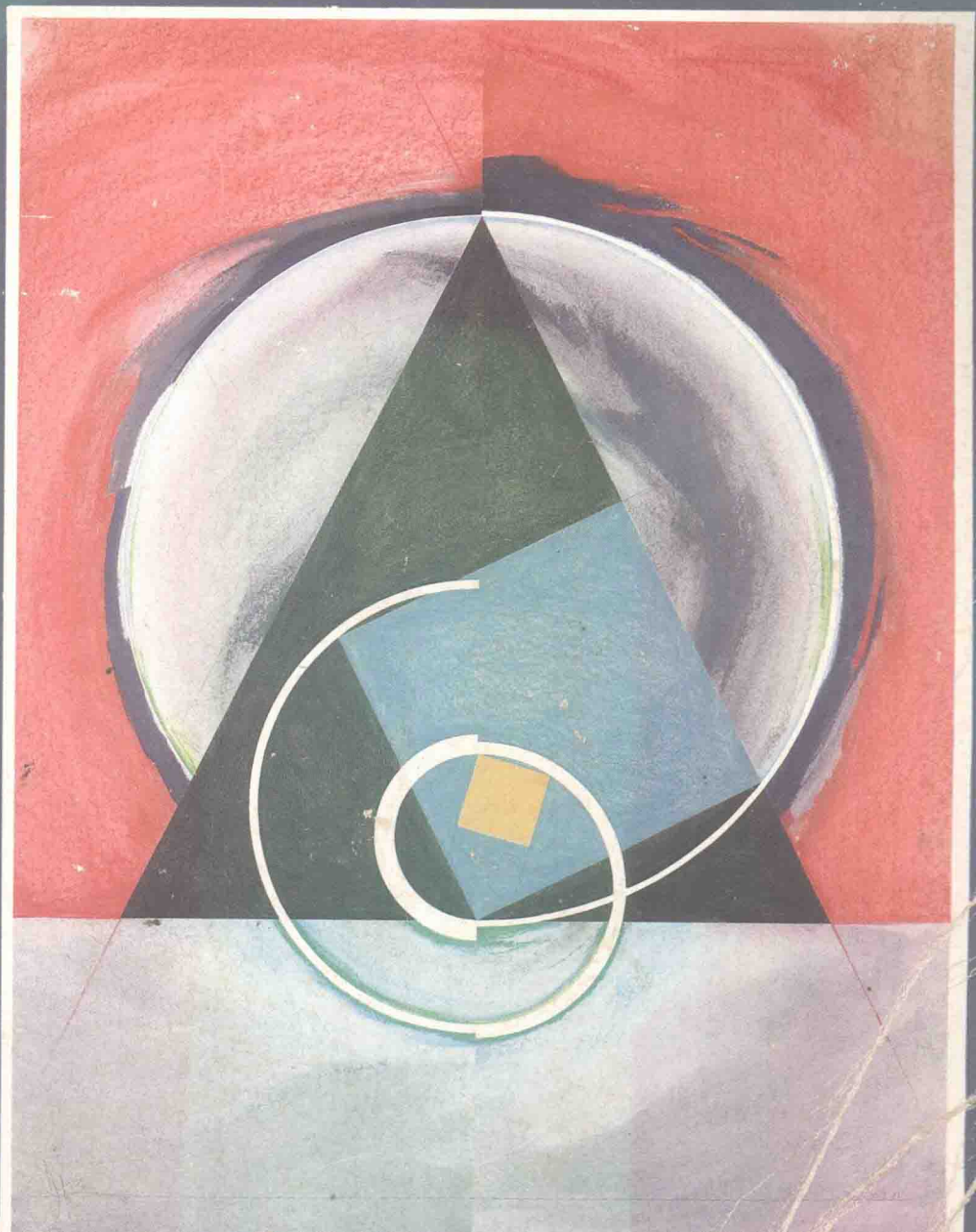
LISP, Objects, and Symbolic Programming

Robert R. Kessler



LISP

Objects, and Symbolic Programming

Robert R. Kessler
University of Utah

With the Assistance of Amy R. Petajan

SCOTT, FORESMAN/LITTLE, BROWN COLLEGE DIVISION
Scott, Foresman and Company
Glenview, Illinois Boston London

To Julie,

for your patience and understanding

Library of Congress Cataloging-in-Publication Data
Kessler, Robert R.
LISP, objects, and symbolic programming / Robert R. Kessler with the assistance of Amy R. Petajan.
p. cm.
Bibliography: p.
Includes Index.
ISBN (invalid) 0-673-39773-1
1. LISP (Computer program language) I. Petajan, Amy R.
II. Title.
QA76.73.L23K47 1988
005.13'3–dc19 87-32312
CIP

1 2 3 4 5 6 7 8 9 10 – RRC – 93 92 91 90 89 88 87

Printed in the United States of America

PREFACE

THIS TEXTBOOK IS for students who have had some exposure to programming. However, it doesn't presume prior LISP experience. This book was derived from the course notes developed for a number of different audiences. Several one-quarter courses at the University of Utah used the notes. The students were mainly upper-division computer science undergraduate and graduate students. There also was an intensive, one-week version of the course taught for industrial companies. Those students were experienced computing professionals with no background in LISP. Although one week is too short to cover this material completely, it provides an adequate foundation of terms and concepts for independent study.

The textbook is structured around case studies of general symbolic computing and artificial intelligence (AI). This textbook is not an in-depth study of AI but provides examples of some algorithms and techniques from that field of computer science. Although you may not be directly involved in AI, the ideas will help you in writing applications in LISP. The chapters before a case study discuss the techniques and features of LISP necessary to solve the case study. Chapter examples are often taken from the case study itself. The chapter discussing the case study doesn't introduce any new LISP programming constructs but demonstrates techniques for developing and interfacing LISP software. It also shows the steps to solve the problem defined by the case study. The discussion of good LISP programming practice is important in each case study chapter.

LISP is a highly interactive language, so you need a running system to learn it. You should spend time directly interacting with a LISP system. Try the various examples to make sure you fully understand the concepts. Determine answers for the exercises and try them out on a running LISP. If you don't understand something, sit down and write some experimental code fragments. Close interaction with a LISP system is the best way to become a good LISP programmer. *All* of the example programs have been run on an existing Common LISP system, specifically Hewlett-Packard Common LISP running on a HP 9000 Model 350 AI workstation. They should all run on any Common LISP.

In the years that I have taught this class to both students in academia and industry, I have observed an interesting phenomenon. Students in academia with a traditional computer science background are comfortable with "black boxes." For example, they think of compilers as software that makes their programs run faster, or "and" gates as performing a particular hardware operation. They are not concerned with the internals of the compiler or with the transistors that make up an "and" gate. Those students should cover the chapters in sequential order. Students with industrial experience, especially those with backgrounds in assembly language, like to see the internals of everything. For those students, the first few sections of Chapter 13 should be covered early. These sections discuss the internal representation of list structures and help you visualize the method in the various LISP functions. Appendix B provides answers to selected exercises. An asterisk before the exercise number in the text indicates that the answer is included in the appendix. The complete set of answers can be found in the solutions manual.

Students with some prior knowledge of LISP might want to skip to Chapter 4, the algebra system case study. An instructor might want to leave the case studies for independent study. However, the salient programming style features should be discussed in class. An instructor might also want to cover the debugging sections of Chapter 10 early on, to help the students as they develop their LISP programs. Chapter 14 covers various LISP programming style rules and should be consulted often during the reading of the text.

A number of people have helped to make this book possible. First and foremost is Tom Casson, executive editor at Scott Foresman/Little, Brown College Division. He convinced me that the book needed to be written and has continually helped push it to its conclusion. Although Amy Petajan only recently became involved in the project, she has helped to improve the presentation immensely. Many members of the Utah Portable AI Support Systems project have reviewed parts of the manuscript. Specifically, Harold Carr, Jed Krohnfeldt, Stan Shebs and Lane Stevens have helped review the technical aspects of the text. Stan has been especially helpful in all phases of the text, including writing the code for the compiler in Chapter 18. Eric Muehle helped to develop many exercises and provided their solutions. Tony Hearn of the Rand Corporation helped with aspects of the Algebra package. Wayne Anderson of the Los Alamos National Laboratory, Martin Griss and Mike Lemon of Hewlett-Packard Research Labs, and Mark Ring of the University of Texas at Austin provided valuable comments to early drafts. Loretta Looser helped in proofreading many of the chapters. The students in CS 546, Autumn 1986, at the University of Utah provided many editing suggestions and answers to some exercises. Thanks to Gerald Maguire, Jr., of Columbia University and John C. Peterson of the University of Arizona for their feedback on early drafts used in their classes. The reviewers of the book, Jeffrey Bonar, Peter Buneman, Douglas Dankel III, Stan Kwasny, Michael Lebowitz, James Meehan, and Margaret P. Mize are gratefully acknowledged. I appreciate the book title suggestions, both serious and humor-

ous, from Wayne Anderson, Mark Bradakis, Al Davis, Martin Griss, Tony Hearn, Micke Hucka, Mike Lemon, Gary Lindstrom, Kwan-Liu Ma, Chip Maguire, Julian Padget, John W. Peterson, Mohammad Pourheidari, Shane Robison, and Leigh Stoller. I also would like to thank Professor Donald Knuth for TEX and Leslie Lamport for LATEX; they have constructed a great document formatting system. I also would like to thank Nelson Beebe of the University of Utah for his help with some of the trickier LATEX commands. I appreciate Hewlett-Packard for donating the many AI workstations, one of which was used to process this book and also as a LISP engine to execute all of the code. Finally, I would like to thank my wife, Julie, for putting up with the long evenings and weekends of writing "the book" during the past two years.

CONTENTS

1 INTRODUCTION TO LISP

JOHN MCCARTHY [27] developed the LISP programming language in the late 1950s. It is the second oldest high-level programming language still in use (FORTRAN is the oldest). LISP is gaining popularity because people are interested in application areas, such as artificial intelligence (AI), requiring the power of LISP. Hardware and software capable of running LISP efficiently is more available as well. As described by Alan Kay [21], programming languages have evolved in distinct generations. LISP has evolved with them. It has been a high-level (FORTRAN, ALGOL), very high-level (SIMULA, PROLOG), and ultra high-level language (VISICALC, EURISKO). This feature has allowed a thirty-year-old language to remain viable.

Due to LISP's long history, many prominent AI programs have been developed in LISP. You can see the history and pragmatics of these implementations by studying the language. You should also study PROLOG [11], another AI language, since it has important features. PROLOG is useful when the control representation is not explicit in the problem, like in data base applications and natural language parsing. LISP makes the control information explicit. This is more natural to programmers familiar with imperative programming languages. There are intensive research efforts to merge the two languages because of significant overlap between them.

Rapid prototyping and program development is a benefit of LISP. Some of the features that allow this are:

- The programmer doesn't need to deal explicitly with pointers, because LISP lets you define and manage flexible, hierarchical data structures;
- The development of systems programs (compilers, editors, etc.) is easy, because LISP programs are represented as LISP data;
- The programmer can manipulate symbolic expressions as well as numeric expressions.
- LISP uses run-time type checking, so the programmer doesn't need to worry about type declarations at compile-time.
- LISP lets you express programs as collections of many small functions, leading to more maintainable software;

- You can develop software in a highly interactive environment; and
- The programmer can strongly customize his or her programming environment to improve software development time.

The following sections describe each of these features in detail. Case studies illustrate them in the rest of the text.

1.1 Data Structures

LISP is an acronym for LISt Processing. The *list* is a flexible, hierarchical data structure that represents arbitrarily long collections of items (for example, numbers and other lists). A list is like those that we use in our everyday lives (lists of things to do, grocery lists, etc.). They grow and shrink as we add and remove items from them. As long as there is enough paper and ink, the lists can be arbitrarily long. Lists can be hierarchical since they can contain other lists (for example, a grocery list referencing all the ingredients in banana bread).

The LISP system has a dynamic storage manager that frees the programmer from worrying about sizes of data structures. In other programming languages, a programmer might allocate a fixed-length vector for a set of values and characteristics of an object. Adding a new value and characteristic is then a problem. The programmer would have to increase the size of the vector for the new characteristic. If a list representation is used, a LISP programmer simply adds the new information to the original list with no other changes necessary. Lists can grow and shrink as necessary, and it's up to the dynamic storage manager to make storage available. Programming languages like Pascal permit this type of structure, but the management of the memory is left to the programmer.

1.2 LISP Programs as LISP Data

LISP programs are collections of user-defined functions. LISP represents them in the same way that it represents data; lists are used to contain both programs and data. This adds a consistency not present in any other language.

Functions that manipulate lists don't distinguish between lists that are functions and those that are data. With this consistency between programs and data, the programmer can edit a function definition from within LISP. For example, suppose that a function has been defined with a variable named `FOOBAR`. Suppose one of the variable references within the function uses the name `FOBAR` (a typographical error). A LISP substitution function could redefine this function, replacing the instance of `FOBAR` with `FOOBAR`.

It is also easy to write system programs (like compilers, editors, and the LISP system itself) in LISP. So there is little distinction between a LISP application programmer and systems programmer. This has helped LISP sys-

tems to evolve, since there is no need for extended communication between the two.

1.3 Symbolic Expressions

A programmer can develop software in LISP that manipulates symbolic expressions, which are collections of words and numbers. Suppose that we wished to describe a clown. For this example, we choose the following subset of characteristics:

Characteristic	Value
shoe size	27
nose	red
expression	frown
name	Freddy

The format of the preceding table shows the relationships between each characteristic and value. The grouping of parentheses in LISP indicates the same relationships:

```
((shoe-size 27) (nose red) (expression frown) (name Freddy))
```

To describe Freddy in a traditional programming language, you might map each characteristic to a numeric value. Thus, the nose color might be a 1 to represent blue, a 2 to represent yellow, a 3 to represent red, and a 4 to represent green. In LISP, you use the natural representation of the word (the *symbol* `red`) for the color of the clown's nose. This technique can also define rules for mixing primary colors. Using a numeric representation, you would have to say something like "if the first color is a 1 and the second is a 2, then the result is a 4." In LISP, you could say "if the first color is `blue` and the second color is `yellow`, then the result is `green`." Since programmers can manipulate these symbols, they don't have to map an abstract notion of color into a numeric representation.

1.4 Variable Declarations

LISP does not require a type declaration for each variable. LISP variables are actually typeless – the values have types instead. A variable's value may at different times be a number, string, list, or any other LISP data type. A LISP function determines its types of arguments during execution and may dynamically coerce one type into another according to the operation. For example, to add an integer and a real number, LISP converts the integer into a real, performs the addition, and returns the sum as a real. Writing generic functions is easy with this dynamic determination and propagation of arbitrary types. You

don't need type-specific functions for each combination of argument types. In a strongly typed language like Pascal, if the same function can accept either an integer or a real argument, two separate functions must be written. In LISP, you write generic functions to dynamically determine and coerce data types as necessary.

A lack of variable declarations does have its disadvantages. Type mismatch errors occur at run-time instead of compile-time, forcing the programmer to do extra debugging. Also, run-time type checking adds computational overhead resulting in slower running programs. Optional type declarations and the development of smarter compilers have alleviated this problem. When the programmer requires more efficiency, he or she may add type declarations to LISP variables. These declarations specify that a variable's value will always be a particular type. The compiler then performs some of the checking at compile-time and generates more efficient code. Recent compilers perform some compile-time type analysis by inferring the types by their usage. This technique generates more efficient code without any additional programmer-supplied declarations. Rapid program development is easier when you can transition no declarations and fully declared LISP code. In the early prototyping stages, the programmer can effectively ignore the data types and concentrate on coding a particular algorithm. Explicit declarations (or inferred ones) can be added for faster program execution as the program moves from experimental to production quality.

1.5 Program Development

LISP is amenable to top-down, bottom-up, or other program development styles. Developing code in any of these styles is easier in LISP than in other languages. This development is an evolutionary process. The programming environment is conducive to experimentation and permits "on the fly" program development. Bottom-up development proceeds naturally from a general idea of the program and data structures. Each part is incrementally developed and tested, eventually resulting in a completed program. Top-down development works like in other languages. The LISP programmer uses the same *Edit*, *Compile*, *Link and Load*, *Execute*, and *Debug* cycle. The main difference is in LISP's dynamic error system, which intercepts calls to undefined functions. When the error occurs, the programmer supplies a return value and then continues execution. This eliminates the need for explicit stub functions and can shorten development time.

Bottom-up development begins with writing many very small functions (functions are rarely longer than one screen). Small functions isolate the processing into small manageable modules. The overhead of function calls is typically very small with LISP, and so this style costs little extra. Highly modular code is easier to maintain. Each module contains small conceptual parts of the program. Isolated changes to a module are less likely to affect the rest of the program. This leads to fewer mistakes and higher quality code. The programmer