artificial intelligence programming

eugene charniak
christopher k. riesbeck
drew v. mcdermott
james r. moehan

second edition
ARTIFICIAL INTELLIGENCE PROGRAMMING
SECOND EDITION
This page intentionally left blank
Contents

Preface xi

PART I: LISP PROGRAMMING

1. LISP REVIEW 1
   1.1 Data Structures 1
   1.2 Program Structures 2
   1.3 Primitive Operations on S-expressions 8
   1.4 Tree Structures 10
   1.5 Lists 12
   1.6 Mapping Functions 14
   1.7 LAMBDA Expressions 15
   1.8 Atoms 17
   1.9 Inside Lisp 18
   1.10 Equality 23
   1.11 Local versus free variables 25
   1.12 Lisp style 27
   1.13 Keywords 29
   1.14 More lambda-list keywords 32

2. MACROS AND READ-MACROS 33
   2.1 Read-Macros 33
   2.2 The Backquote Read-Macro 34
   2.3 Terminating read-macros 39
Contents

2.4 Macros 39
2.5 Generating new symbols 45
2.6 BIND: A Macro For Special Variables 47

3. DATA STRUCTURES AND CONTROL STRUCTURES IN LISP 48
  3.1 The Need for Data Types 48
  3.2 The Conservative Approach to Type Definition 50
  3.3 The Liberal Approach 51
  3.4 The Radical Approach 53
  3.5 Control Structures 55
  3.6 Basic control structure 56
  3.7 Local functions: FLET and LABELS 56
  3.8 Interrupting the normal flow 58
  3.9 Redirecting the flow of control: GO 59
  3.10 ITERATE 60
  3.11 DO 63
  3.12 Iteration and lists: the mapping functions 64
  3.13 Defining the FOR-macro 69
  3.14 CATCH and THROW 74
  3.15 UNWIND-PROTECT 74
  3.16 BIND 75
  3.17 Conclusion 82

4. INPUT/OUTPUT IN LISP 83
  4.1 Streams 84
  4.2 READ-CHAR, READ-LINE, and READ 85
  4.3 PRIN1, PRINC, and TERPRI 88
  4.4 PRINT, PPRINT, and FORMAT 88
  4.5 The MSG Macro 90
  4.6 Separating I/O from Your Functions 93

5. COMPILING YOUR PROGRAM AND YOUR PROGRAM'S PROGRAM 97
  5.1 What Is Compilation? 97
  5.2 Implications for AI Programs 99
  5.3 Example: Regular Expressions 100
  5.4 What the Lisp Compiler Does 101
  5.5 Compiler Declarations 105
  5.6 Macros in Compiled Code 107
  5.7 Variables in Compiled Code 107
5.8 Lexical Scoping Versus EVAL and SET 108
5.9 Ignored Variables 110

6. **DATA-DRIVEN PROGRAMMING AND OTHER PROGRAMMING TECHNIQUES** 111
   6.1 Data-driven Programming 111
   6.2 Association Lists, Property Lists, and Hash Tables 112
   6.3 Reimplementing MSG 116
   6.4 Data-Driven Programming as an Organizational Device 119
   6.5 Set Operations on Lists 121
   6.6 Headed Lists and Queues 123

7. **HIGHER-ORDER FUNCTIONS, CONTINUATIONS, AND COROUTINES** 127
   7.1 Passing Procedures In and Out 127
   7.2 Continuations and Tail-Recursion 129
   7.3 Continuations and Multiple Values 132
   7.4 Coroutines 136
   7.5 Continuations and Control Flow: CALL-WITH-CURRENT-CONTINUATION 142
   7.6 Problems with Continuation Passing 147

**PART II: AI PROGRAMMING TECHNIQUES**

8. **SIMPLE DISCRIMINATION NETS** 149
   8.1 The General Discrimination Net 149
   8.2 Database Discrimination Nets—Lists of Atoms 153
   8.3 Database Discrimination Nets—General S-expressions 157
   8.4 Implementing Discrimination Trees with Continuations 161

9. **AGENDA CONTROL STRUCTURES** 166
   9.1 Introduction 166
   9.2 Best-First Tree Search 166
   9.3 Coroutines and Agendas 171
   9.4 Design Alternatives for Agendas 176
   9.5 Generated Lists 177

10. **DEDUCTIVE INFORMATION RETRIEVAL** 182
    10.1 Introduction 182
10.2 Database-defined Predicates 183
10.3 Connectives, Variables, and Inference Rules 183
10.4 Existential Quantification 187
10.5 Coming to Terms 188
10.6 Issues 189
10.7 Deductive Retrieval 190
10.8 A Unification Algorithm 195
10.9 A Deductive Retriever 202
10.10 Forward Chaining 213
10.11 Programming with Logic 215
10.12 Extending the Retriever 218
10.13 Deductive Retrieval versus Theorem Proving 223
10.14 Pattern Matching 224
10.15 The Pros and Cons of Deductive Information Retrieval 226

11. DISCRIMINATION NETS WITH VARIABLES 230
11.1 Plan Retrieval 231
11.2 Fetching Facts 235
11.3 Variations on the Discrimination Net Theme 240

12. PRODUCTION SYSTEMS 248
12.1 Representation of Rules in XPS 249
12.2 Representation of Working Memory Elements in XPS 250
12.3 Establishing the Conflict Set 251
12.4 Conflict Resolution in XPS 251
12.5 The Implementation of XPS 254
12.6 An Example 266
12.7 Improvements to XPS 271
12.8 XPS and OPS5 272
12.9 Deductive Retrieval and Production Systems 275

13. SLOT AND FILLER DATA BASES 276
13.1 Expanding Property List Facilities 276
13.2 An Introduction to XRL 277
13.3 IF-ADDED Methods 287
13.4 Pattern Matching in XRL 288
13.5 Indexing Forms 294
13.6 Retrieving Forms 296
13.7 Extending XRL 298
Contents

14. CHRONOLOGICAL BACKTRACKING 304
   14.1 Introduction 304
   14.2 A Basic Transition Network Grammar without Backup 305
   14.3 Representing an ATN Network in Lisp 308
   14.4 An ATN without Backtracking 312
   14.5 Backtracking In ATNs—A State-Saving Approach 320
   14.6 Backtracking with a Transition-Saving Approach 327
   14.7 Defining the ATN with Continuations 328
   14.8 Using Problem-Solving Techniques 332

15. DATA DEPENDENCIES AND REASON MAINTENANCE SYSTEMS 337
   15.1 The Need for Reason Maintenance 337
   15.2 Data-Dependency Clauses 338
   15.3 Data-Dependency Network Graphs 339
   15.4 Labeling Data-Dependency Networks 339
   15.5 Propagating Labels from New Clauses 341
   15.6 Incompleteness of the RMS Algorithm 343
   15.7 retracting Clauses and Delabeling Nodes 345
   15.8 The Implementation of an RMS 347
   15.9 The Eight Queens Problem 355
   15.10 Dependency-directed Backtracking 367
   15.11 Finding Nogoods 372
   15.12 Subsuming Clauses 378
   15.13 Dealing with Interruptions 379
   15.14 RMS to Lisp Communications 385

Appendix 1: A Glossary of Common Lisp Functions 389
Answers to Selected Exercises 441
Bibliography 495
Author Index 505
Index of Defined LISP Items 507
Subject Index 515
This page intentionally left blank
Since the first edition of this book appeared, some things in the AI programming world have changed a great deal, and some things are almost exactly the way they were five years ago. Perhaps the most significant development has been the appearance of COMMON LISP, documented in abundant detail by Guy Steele [102]. All of the LISP code in this new edition has been rewritten in COMMON LISP. COMMON LISP is a pleasant surprise, given the normal result of compromise solutions designed by committees. It manages to be a synthesis of many of the best ideas present in modern LISP dialects, rather than a fossilization of the worst. While there are other dialects that have a more coherent semantics, such as SCHEME [82] and T [81], COMMON LISP is more than adequate for our needs.

The choice of COMMON LISP has affected the contents of this book in several ways. First, many features that we spent some time developing and adding to our earlier dialect of LISP are already available in COMMON LISP. In some cases, we have therefore just described the COMMON LISP feature. In other cases, we have retained the developmental material in order to explain the underlying principles.

COMMON LISP is a "large" language, and we cover only part of it; some of the best-designed features allowed us to remove material from the first edition that dealt with the friendly but limited dialect we used then, UCI LISP. Gone are the sections on FEXPRs and LEXPRs; while we still discuss the issue of extending the language by adding new data types, COMMON LISP's DEFSTRUCT is an example of a tool that we had to build from scratch in UCI LISP.

We considered both SCHEME and T for this edition. SCHEME has the right essential semantics for programming, such as lexical scoping and closures, and
T extends that to include the right primitives for object-oriented programming and language-extension via macros, as well as a host of well-designed support features. (The next edition (!) will probably be in T.) Happily, many of these ideas are also present in COMMON LISP, and given its greater visibility, we chose it instead. In particular, the availability of lexical closures has allowed us to re-implement a number of disparate ideas in a uniform manner.

One problem with COMMON LISP is that it is not yet widely available. Most implementations exist only on very large machines. While this situation will certainly change in the next few years, we have tried to ameliorate the problem in two ways. First, we have used only a subset of COMMON LISP in our code. Second, we have provided a glossary describing the subset we used.

In the LISP chapters in Part 1 and in the AI topics in Part 2, five years’ additional experience has led us to provide completely new explanations, examples, and implementations. The original chapter on alternative control structures, which described the implementation of a variation of SCHEME, served two purposes: to introduce some of the power behind lexically scoped languages with procedures as first-class objects, and to give an example of how other languages with very different control structures could be implemented in LISP. Since COMMON LISP is lexically scoped, there is no longer any need to treat that topic separately. The topic of control structure in SCHEME is now described briefly in a new chapter on higher-order functions, continuations, and coroutines. Some of the flavor of implementing interpreters can be gleaned from a new chapter on production systems. (For more information on SCHEME and implementing a SCHEME interpreter, the interested reader is referred to Abelson and Sussman [2].)

The chapter on production systems illustrates several aspects of AI programming. A production system is much easier to implement than a deductive retriever, and correspondingly more limited in power. Production systems, however, have proved to be very useful in the development of expert systems, which are the basis for most of the commercial AI work at the moment.

As before, the intended audience of Artificial Intelligence Programming remains the advanced undergraduate or early graduate student in AI. Although the material involved requires only modest knowledge of programming, the student who has had the most experience in creating AI programs already will understand best the benefits of the techniques described. In addition, in response to the changing nature of AI in industry, we have changed the text to be a bit more like a cookbook. It has turned out that many people learning AI programming are doing so on their own, either at home or at work. Many of them prefer to begin with working pieces of code that they can then extend, rather than inadequate pieces of code that are then corrected in the text and exercises. This latter technique works in classrooms, but not in a self-study situation. Therefore, we have eliminated almost all figures with deliberately incorrect code, and added an appendix with the answers to nearly all the exercises in the book. We have also removed the chapters on the sample course project.
ACKNOWLEDGMENTS

We would like to thank the teachers, students, and other programmers who suggested changes (and supplied corrections!) to the first edition, as well as reviewers who gave us criticisms prior to this revision. Thanks to Paul Hudak and Bill Ferguson for comments on Chapter 7, and to Robert Farrell for comments on Chapter 12. We are indebted to Bob Strong at Cognitive Systems Inc. for reviewing the new edition, recommending improvements, and helping us test the code. Answers to some of the exercises were written by Gregory Parkinson and Rika Yoshii when they were students at UCI. The Yale Artificial Intelligence Project and Cognitive Systems Inc. provided computer support for the preparation of the new manuscript. Finally, Chris Riesbeck would like to dedicate this book to Maxine.

Christopher K. Riesbeck
Jim Meehan

PREFACE TO THE FIRST EDITION

Artificial Intelligence (henceforth AI) is still a field where disagreement is more common than solid theory, and interesting ideas more common than polished programs. Yet there is slowly coming into being a small core of accepted (though not universally accepted) theory and practice. This book is an attempt to gather together the "practice" aspect of this "core" AI.

The "practice" of AI is, of course, the writing of programs. AI problems are usually ill-defined and the theories proposed are often too complex and complicated to be verified by intuitive or formal arguments. Sometimes the only way to understand and evaluate a theory is to see what comes next. To find this out, and to check for obvious inconsistencies and contradictions, we write programs that are intended to reflect our theories. If these programs work, our theories are not proved, of course, but at least we gain some understanding of how they behave. When the programs don't work (or we find ourselves unable to program the theories at all), then we learn what we have yet to define or re-define.

With this emphasis on programming, it becomes important that an AI researcher have a wide library of programming tools available. This is particularly true because of the "level" problem. That is, your theory describes what to do at a fairly high level, but you need to tell the machine what to do at a low level. So a theory of, say, coherency in conversation, will in all probability say nothing about pattern matching or efficient data retrieval. It is not that these topics are not worthy of their own theory. How people manage to retrieve knowledge efficiently under a wide variety of circumstances is a fascinating
question. But if you are worried about conversation, it is simply not your department.

The intent of this book is to give you a wide variety of commonly used tools for programming Artificial Intelligence theories: discrimination nets, agendas, deduction, data dependencies, backtracking, etc. By having these tools, we hope you will find that your programs better reflect your intentions.

Almost all of the ideas that are described here are in common use, particularly at the larger Artificial Intelligence centers. But very few of them have ever been written down in one place. There are a number of books that introduce you to Lisp (although none of them are completely satisfactory), and there are a number of books on theories and algorithms in Artificial Intelligence. Until now, however, there have been no books that fill in the middle ground and present the methods that all the old-timers know for getting from theory to practice. That is what this book is all about.

The major problem in writing a book such as this is that of selection. In some cases it is easy. It seems unlikely that anyone would seriously contest our inclusion of discrimination nets, pattern matching, or agendas. These techniques have been used by many researchers in the field and in a variety of problem areas—from natural language comprehension to problem solving to medical diagnosis. However once one moves beyond this handful of topics, or even starts getting specific about the type of pattern matching, or agenda, then consensus is not so easy. So to some degree the selections made in this book are personal ones. Of course, to say they are personal is not to say they cannot be defended on scientific grounds, but rather that the defense would take the form of an extended debate on the nature of AI and where it is going. For example, data dependencies receive a chapter to themselves here in spite of the fact that they are fairly new on the scene and hence relatively untested, at least compared to something like unification pattern matching. Naturally we try to show the usefulness of these ideas, but only to show how the ideas are motivated, not to defend particular approaches against competitors. Such a defense would be well worth having, but it would be out of place in a text such as this.

Selection also implies that some things are omitted, and there are at least two notable omissions from these chapters. One of these is inadvertent. The techniques discussed here all come from what might be thought of as "abstract" AI. That is, if we think of AI programs on a spectrum from "concrete" programs which must deal with the real world in terms of sound and light input (or sound and muscle output) to "abstract" programs which only deal with abstractions, the techniques described here fall most naturally towards the abstract end. This book does not have the space and the authors do not have the expertise to do justice to the concrete end of things.

A second omission is quite deliberate. We have made no attempt to survey, much less teach, the many AI languages (Conniver, QA-4, KRL, etc.). This stems from our conviction that at present there is no commonly agreed-upon set
of functions above the level of list processing which everyone would agree is useful in a wide variety of AI settings. Experience has shown that each major project has found it necessary to build up its own tools, starting, typically, from LISP. We do not see this situation as likely to change in the foreseeable future. Hence rather than covering the basics of the various languages we have tried instead to explain the techniques which typically lie behind these languages.

The book is divided into two parts. Since almost all serious programming in Artificial Intelligence is done in the language LISP, Part 1 tells you how to improve your general abilities as a LISP programmer. [The first chapter covers] most of the basic LISP concepts needed for the rest of the book. We intend the introductory material to cover all the concepts of LISP needed later, but if you have never programmed in LISP before, we recommend that you spend some time writing simple LISP programs until you get a feel for the language.

The [second through seventh] chapters are concerned with the many features found (or implementable) in LISP that make the language an attractive one to use. Many of the ideas that pass under the rubric of "structured programming" will be found here. Although LISP is almost as old as FORTRAN, it is surprisingly amenable to things like top-down programming and data types.

Part 2 contains more advanced and complex techniques. Since this book is intended not just to be a description of ideas, but also to give you a chance to learn the craft of Artificial Intelligence, we present actual LISP implementations of all the ideas discussed, along with exercises which modify and extend the code. These exercises are intended to make you familiar, in a practical hands-on way, with the techniques involved. We hope that the exercises will inspire you to experiment and learn on your own.

This book is intended mainly for use as a textbook for an AI course in which programming is emphasized. This could be either an advanced or a fast elementary course. The book might also be used as an auxiliary text for a systems course; for this purpose, the chapters on macros, structured programming, and alternative control structures would be most useful.

ACKNOWLEDGMENTS

The first edition of this book was the outgrowth of a graduate course given by the authors in the spring of 1978 at Yale University for students in the Artificial Intelligence Project of the Yale Computer Science Department. We would like to acknowledge Roger Schank who originated the idea of the course. We'd also like to thank Dave Barstow and Walter Stutzman for their detailed comments on the complete manuscript, Laury Miller and Glenn Edelson for their comments on the introductory chapters, and Jon Doyle for his comments on the chapter on data dependencies.

The Yale Artificial Intelligence Project is funded by the Advanced Research Projects Agency of the Department of Defense and the Office of Naval Research.
During the writing of [the first edition of] this book, Eugene Charniak and Christopher Riesbeck were supported by the Advanced Research Projects Agency monitored under the Office of Naval Research under contract N00014–75–C–1111.

Eugene Charniak
Christopher K. Riesbeck
Drew V. McDermott
LISP PROGRAMMING
This page intentionally left blank
Lisp has jokingly been called "the most intelligent way to misuse a computer." I think that description is a great compliment because it transmits the full flavor of liberation: it has assisted a number of our most gifted fellow humans in thinking previously impossible thoughts.

— Edsger Dijkstra

LISP was the world's first elegant language, in the sense that it provided a parsimonious base with rich possibilities for extension. LISP has been applied mainly to problems of symbolic manipulation and artificial intelligence, partly because manipulating symbols is so easy in LISP, and partly because AI programmers tend to be lazy and undisciplined, like pilots who refuse to file a flight plan before taking off, and LISP's interactive structure allows them to get away with this.

1.1 Data Structures

LISP data structures are called "S-expressions." The S stands for "symbolic." In this text, the terms "S-expression" and "expression" are used interchangeably. An S-expression is

1. a number, e.g., 15, written as an optional plus or minus sign, followed by one or more digits.
2. a symbol, e.g., NODE, written as a letter followed by zero or more letters or digits.
3. a *string*, e.g., "This is a string", written as a double quote, followed by zero or more characters, followed by another double quote.
4. a *character*, e.g., #\'q, written as a sharp sign, followed by a backslash, followed by a character. (Numbers, symbols, strings, and characters are called *atoms*. There are other kinds of atoms. We will see them in later chapters.)
5. a *list* of *S*-expressions, e.g., (A B) or (IS TALL (FATHER BILL)), written as a left parenthesis, followed by zero or more *S*-expressions, followed by a right parenthesis.

Parentheses are more significant in *Lisp* than they are in most other programming languages. Parentheses are virtually the only punctuation marks available in *Lisp* programs. They are used to indicate the structure of *S*-expressions. For example, (A) is a list of one element, the symbol A. ( (A) ) is also a list of one element, which is in turn a list of one element, which is the symbol A. Notice also that the left and right parentheses must *balance*. That is, a well-formed *S*-expression has a right parenthesis to close off each left parenthesis.

1.2 Program Structures

**Syntax**: The syntax of a *Lisp* program is simple: Every *S*-expression is a syntactically legal program! That is, any given data structure could be executed as a program. Most of them, however, fail on semantic grounds.

**Semantics**: The function that executes *S*-expressions (and hence defines the semantics of *Lisp*) is called **EVAL**. **EVAL** takes one *S*-expression and returns another *S*-expression. The second expression is called the *value* of the first expression. We notate this as *expression* \(\Rightarrow\) *value*.

The rules for evaluation are fairly simple.

**Rule 1**: If the expression is a *number*, a *string*, a *character*, the symbol T, or the symbol NIL, then its value is itself. So 5 \(\Rightarrow\) 5.

**Rule 2**: If the expression is a *list* of the form

\[(function\ arg_1 \ldots \ arg_n)\]

then the value is found by first evaluating each argument (*arg* 1 to *arg* n), and then calling *function* with these values. For the moment, all our functions are named by symbols. For example, the symbols + (for addition) and * (for multiplication) name functions that are defined in *Lisp* for doing arithmetic.

\[
\begin{align*}
(+\ 15\ 2) &\Rightarrow 17 \\
(*\ 3\ 5) &\Rightarrow 15 \\
(+\ (*\ 3\ 5)\ 2) &\Rightarrow 17
\end{align*}
\]
Note that in order to evaluate the last expression, each argument has to be evaluated first. Since the first argument is itself a list, \((\ast 3 5)\), **Rule 2** is applied again, and the arguments, 3 and 5, are evaluated. By **Rule 1**, they evaluate to themselves. They are passed to the multiplication procedure, \(*\), which returns 15. Similarly, 2 evaluates to itself, and 15 and 2 are passed to \(+\), which returns the number 17.

**Rule 3**: If the expression is a list of the form

\[(\text{reserved-word} \ \text{arg}_1 \ldots \text{arg}_k)\]

then the value depends completely on the definition of \text{reserved-word}. The arguments may or may not be evaluated. Reserved words are named by symbols, just as functions are.

One such reserved word is \texttt{SETQ}. \texttt{SETQ} is used for assigning values to symbols. \((\text{SETQ} \ \text{symbol} \ \text{expression})\) causes \text{symbol} to be "bound to" (or "set to" or "assigned") the value of \text{expression}. The value of \text{expression} is returned as the value of the \texttt{SETQ} (i.e., the value that the \texttt{SETQ}-expression produces).

For example, \((\text{SETQ} \ x \ (\text{+} \ 15 \ 1))\) sets \texttt{x} to 16.

**Rule 4**: If the expression is a symbol, then its value is the last value that has been assigned to it. If no value has been assigned, then an error occurs. So if \texttt{x} is bound to 16, then \(x \Rightarrow 16\). These are the four rules employed by the EVAL, which is part of the LISP interpreter. The interpreter is a program that you run. You type an S-expression and it prints back another S-expression. This second S-expression is the value of the one you typed in. (If something goes wrong, an error message is printed instead of a value).

LISP programs don’t always need to be compiled like ALGOL or FORTRAN programs. That is, you do not have to take a file of LISP text, convert it into internal machine code, and then run the machine code. Instead, you can type LISP text to the interpreter, which evaluates it and types the result back at you as more LISP text.

While interpreting means that programs run slower, it also usually means that you always have your expressions available during execution for inspection and modification. LISP can support very powerful debugging and editing facilities for this reason.

The "top-level" loop of the LISP interpreter can be written in a ALGOL-like language as

\[\begin{align*}
\text{BEGIN} \\
\text{LOOP: EXP := READ (INPUT);} \\
\quad \text{VAL := EVAL (EXP);} \\
\quad \text{PRINT (VAL, OUTPUT);} \\
\quad \text{GO TO LOOP}
\end{align*}\]

This is usually referred to as the **READ-EVAL-PRINT** loop. All three functions are available to the user — that is, when you write a function that uses
the LISP functions READ, EVAL, or PRINT, you are calling the same machine code that the interpreter uses in the READ-EVAL-PRINT loop.

The function READ reads one S-expression. It knows about balanced parentheses and the spaces that terminate numbers and atoms. It takes one argument, a stream, which is a LISP object that indicates the source of the input.

The function EVAL applies the four rules of evaluation to its argument and returns a new S-expression unless an error occurs, in which case an error message is printed.

The function PRINT takes two arguments: an S-expression, which it prints; and a stream, which indicates where the output should be sent. Internally, LISP stores everything as machine addresses, not as character strings. The functions READ and PRINT are responsible for converting strings with letters, numbers, spaces, and parentheses into list structures and back again. It is important to realize that what you type and what LISP reads are character strings, not LISP lists.

For example, every time you type the name of a LISP function like PRINT, you want it to refer to the same internal machine code for printing. The function READ is responsible for taking the characters P, R, I, N, and T and converting them into a reference to a unique address. We will see shortly several other jobs that READ does.

The fact that the interpreter reads what you type and prints back its value explains why very few of the functions we define have any explicit input/output references. In LISP, almost everything is done by passing values between functions, including communication from the user to the interpreter and back.

Furthermore, in LISP, you do not define a "main program" with subroutines. Instead you define a set of functions. One of them may be the only one that you intend to call explicitly at the top level of LISP, but this does not rule out calling any of the others if you want to. All functions are equal in the sight of the LISP interpreter.

Learning LISP is a matter of learning the reserved words and built-in functions, plus the appropriate programming techniques. A list of the reserved words and functions needed for this book appears in the glossary.

When we describe the various functions and reserved words, we need an informal notation for giving the syntax of the LISP expressions they are called with. The following conventions are used:

1. Any word in lowercase italics stands for an arbitrary LISP expression which can be substituted where the word appears. Usually the word suggests what kind of expressions can be used (e.g., atom or expression).
2. Anything that is enclosed in braces and followed by an asterisk stands for zero or more occurrences of that thing, separated by spaces (e.g., {atom}*).
3. Items enclosed in braces and separated by a vertical bar indicate a choice.
For example, `(FLET | LABELS)` means "either FLET or LABELS."

4. Anything else is exactly what should appear in the LISP expression.

Figure 1.1 contains an example of what a LISP expression looks like. It is also a valid LISP program.

```lisp
(LET ((SUM 0))
  (LOOP (IF (EQUAL N 0) (RETURN SUM))
     (SETQ SUM (+ SUM N))
     (SETQ N (- N 1))))
```

FIGURE 1.1 Sample LISP expression

The READ function ignores extra spaces and ends of lines. Therefore, you can (and should) break up expressions over several lines, with a great deal of indentation to set things off, in order to make it easier for you and other programmers to read. There are special printing functions that do this for you automatically.

The program in Figure 1.1 adds up the integers from 1 to N. The reserved words used are LET, LOOP, IF, RETURN, and SETQ. The functions used are EQUAL, +, and -.

LET-expressions are treated like BEGIN/END blocks in ALGOL. The first expression after the LET symbol is a list of pairs (2-element lists). The first item in each pair is a symbol that names a local variable, and the second item is the initial value assigned to that variable. The previous values assigned to these variables are saved. When the evaluation of the LET is finished, these previous values are restored, and whatever values were assigned during the evaluation of the LET are lost.

In this case, the only local variable is SUM. It is bound to 0 when the LET is entered, and re-bound to its previous value when the LET is exited. If it didn't have a previous value, then we say that it was previously "unbound," meaning that it had no value at all, and it becomes unbound once again after exiting the LET.

The rest of the expressions in a LET are called the "body" of the LET. They are normal LISP expressions, and they are evaluated in order. The value of the last of these expressions becomes the value of the LET.

In our example, there is only one LISP expression in the body. It's a LOOP-expression. LOOP is another reserved word, not a function. The expressions inside the LOOP are evaluated in order. After the last expression is
evaluated, they're all evaluated again, and again, forever. The only way out of
a LOOP is to call RETURN.

The first expression inside this LOOP is an "if-then" statement, also known
as a conditional. Its general form looks like this:

\[(\text{IF test consequent alternate})\]

Test, consequent, and alternate are LISP expressions; the alternate is optional;
there isn't one in our example, for instance.

We can read this as, "If test then consequent else alternate." IF starts by
evaluating test. If the result is true, then consequent is evaluated, and its value
becomes the value of the IF-expression. If the result is false, then alternate is
evaluated, and its value becomes the value of the IF-expression. If the result is
false and there isn't an alternate, then the value of the IF-expression is false.

Notice that the if-then-else form in LISP returns a value; in many programming
languages, it does not.

False in LISP is represented by the symbol NIL. True is represented in LISP
by anything that is not NIL. Hence 1, Foo, and (A B C) are all true. The
symbol T is commonly used to represent true. For example, (EQUAL N 0)
returns T if N equals zero, and NIL otherwise.

T and NIL are convenient to use because they are pre-assigned in LISP to
have themselves as their values. That is, the value of T is T, and the value of
NIL is NIL.

(It may take you a while to get used to true being indicated by any non-NIL
expression. Many quasi-predicates like MEMBER take advantage of this broad
definition of true by returning something more useful than a mere T in the case
where they succeed.)

RETURN is a reserved word. It evaluates its argument (SUM in our example),
and then, instead of following the normal rules for evaluation, it breaks out of the
nearest LOOP; the value of its argument becomes the value of that LOOP-
expression.

So if N is zero, then the value of SUM is immediately returned as the value
of the LOOP. Since that's the last (and only!) expression inside the LET-
expression, it also becomes the value that the LET-expression returns.

The other expressions in the LOOP should look familiar by now. We increase
the value of SUM by N, and decrease the value of N by 1. Then because we're
in a LOOP, we start over again with the IF-statement. Eventually, N will be
zero, and SUM will be returned.

Figure 1.2 shows another way of writing the same program.

In this version, we use a different form of conditional expression, COND.
COND is a very elegant construct. Its format is given in Figure 1.3.

Each list inside the COND is called a COND-clause. The first element of each
COND-clause is a test. The tests are evaluated in order until one evaluates to
true, then the expressions in the rest of that test's clause are evaluated. The value
(LET ((SUM 0))
  (LOOP (COND ((EQUAL N 0) (RETURN SUM))
    (T (SETQ SUM (+ SUM N))
      (SETQ N (- N 1))))))

FIGURE 1.2 An alternate version

(COND (test1 {expression}*)
  (test2 {expression}*)
  ...
  (testn {expression}*)

FIGURE 1.3 Format of CONDITIONAL expressions

of the last of those expressions is returned as the value of the whole
COND-expression, and the other clauses are bypassed. Thus, if test1 returns
true, then the expressions following it will be evaluated, and the value of the last
one will be returned. If test1 returns false, then test2 will be evaluated, and so on.

The last test in a COND is often just T. Since T is always true (non-NIL), this
means that if the COND reaches the last branch, that branch will be taken. Thus
the T serves the role of “else” in the conditional.

You may have heard that Lisp is a very strange language, but this example
should demonstrate that it is fundamentally very conventional. Its syntax may
take getting used to, but it has many advantages. By fully parenthesizing
everything, we avoid having to write BEGIN/END or DO/DOD pairs. The
tabular form of conditional means that there is no “dangling ELSE” problem,
and no need to insert clumsy BEGINS and ENDS inside conditional clauses.

We have also seen that there are two ways to do a sequence of operations in
Lisp. We have just looked at LET and LOOP, which allow us to evaluate
expressions in a linear sequence. Earlier we noted that because the evaluation
procedure itself was recursive, evaluating the arguments of one function call
could lead to evaluating other function calls.

For example, the normal LISP way of writing the READ-EVAL-PRINT
loop of the LISP interpreter would be

(LOOP
  (PRINT (EVAL (READ *STANDARD-INPUT*)))
  *STANDARD-OUTPUT*))

Notice that the evaluation is from outside to inside, but that in order to
evaluate the PRINT, LISP first has to evaluate the call to EVAL, and that in
order to evaluate that, LISP first has to evaluate the call to \texttt{READ}. Hence, the actual order of execution becomes \texttt{READ}, then \texttt{EVAL}, and finally \texttt{PRINT}.

1.3 Primitive Operations on S-expressions

Our first example operated on numbers the way any other language would. Lisp is, however, mainly designed for performing symbolic manipulations. Its most important functions are those that apply to symbolic expressions.

Two central functions are \texttt{CAR} and \texttt{CDR}. \texttt{CAR} takes a list and returns the first element of that list. \texttt{CDR} takes a list and returns the rest of the list minus the first element. In some other languages having list structures, \texttt{CAR} and \texttt{CDR} are called \texttt{HEAD} and \texttt{TAIL}, respectively.

With \texttt{CAR} and \texttt{CDR} we can get at any subpart of a list structure. For example, by taking the \texttt{CDR} of a list, followed by the \texttt{CAR} of the result, we get the second element of the original list. By taking two \texttt{CDR}s and then a \texttt{CAR}, we get the third element, and so on.

Assume we have the list \((\texttt{A B C})\). We immediately run into a problem if we try to apply \texttt{CAR} or \texttt{CDR} to this list. Suppose we try writing the following

\[ \texttt{(CAR (A B C))} \]

Compare this with the form

\[ \texttt{( + (* 3 5) 2)} \]

Remember that in evaluating the \texttt{+} expression, LISP first evaluated the \texttt{*} subexpression. The same thing happens with the \texttt{CAR} expression. LISP first tries to evaluate the subexpression \((\texttt{A B C})\). LISP assumes that any atom at the head of a list that is not known to be a reserved word must be a function. Therefore, by Rule 2, it tries to apply the function \texttt{A} to the values of the symbols \texttt{B} and \texttt{C}. This is not what we want. We want \((\texttt{A B C})\) to be left alone.

To do this, we use the reserved word \texttt{QUOTE}. \texttt{QUOTE} returns its one argument unevaluated. That is,

\[ \texttt{(QUOTE expression)} \Rightarrow \texttt{expression} \]

Since \texttt{QUOTE} turns out to be a very commonly needed function, LISP provides an abbreviation. An expression of the form \texttt{'}exp is transformed by the function \texttt{READ} into \texttt{(QUOTE exp)}. Thus to get the list \((\texttt{A B C})\) we can use either

\[ \texttt{(QUOTE (A B C))} \]

or

\[ \texttt{'(A B C)} \]
To get pieces of this list, we write

\[
\begin{align*}
& (\text{CAR} \ (\text{QUOTE} \ (A \ B \ C))) \Rightarrow A \\
& (\text{CDR} \ (\text{QUOTE} \ (A \ B \ C))) \Rightarrow (B \ C) \\
& (\text{CAR} \ (\text{CDR} \ (\text{QUOTE} \ (A \ B \ C)))) \Rightarrow B
\end{align*}
\]

What happens if we take the \text{CDR} of a list of one element? Watch!

\[
(\text{CDR} \ (\text{QUOTE} \ (A))) \Rightarrow ()
\]

The \text{CDR} of a list that has one element is the empty list, which reasonably enough is written \((\)). In Lisp the empty list \((\)) is the value of the symbol \text{NIL}. Thus \text{NIL} stands for the empty list as well as \text{false}.

Given \text{CAR} and \text{CDR}, we can write a \text{LOOP}, analogous to our first one, that sums up the numbers in a list \(L\) of numbers (see Figure 1.4).

\[
(\text{LET} ((\text{SUM} \ 0)) \\
(\text{LOOP} \ (\text{IF} \ (\text{EQUAL} \ L \ \text{NIL}) \ (\text{RETURN} \ \text{SUM})) \\
(\text{SETQ} \ \text{SUM} \ (+ \ \text{SUM} \ (\text{CAR} \ L))) \\
(\text{SETQ} \ L \ (\text{CDR} \ L)))
\]

\textbf{FIGURE 1.4} Expression to add a list of numbers

If \(L\) were assigned the value \((1 \ 2 \ 3 \ 4)\), then this expression would return the value 10.

So far we have seen how we can decompose a list using \text{CAR} and \text{CDR}. Of course, lists have to be built up as well as taken apart. To build them, we use the function \text{CONS}. \((\text{CONS} \ x \ l)\) creates a new list with \(x\) as the first element followed by the elements of \(l\).

For example,

\[
(\text{CONS} \ (\text{QUOTE} \ A) \ (\text{QUOTE} \ (B \ C))) \Rightarrow (A \ B \ C) \\
(\text{CONS} \ (\text{QUOTE} \ B) \ \text{NIL}) \Rightarrow (B)
\]

What happens if we \text{CONS} an atom onto another atom (besides \text{NIL}, which represents the empty list)?

\[
(\text{CONS} \ \text{'A} \ \text{'B}) \Rightarrow (\text{'A} \ \text{'B})
\]

This is called a “dotted pair.” Dotted pairs are actually the fundamental data structures of Lisp, but we delay describing them until later, in the section called “Inside Lisp.”

Note that for any non-empty list, the following identity holds:

\[
(\text{CONS} \ (\text{CAR} \ \text{list}) \ (\text{CDR} \ \text{list})) = \text{list}
\]

There are two other very handy functions for building lists: \text{APPEND} and \text{LIST}. \text{APPEND} takes two or more arguments, all of which should be lists
(empty lists are allowed), and returns a list built from the elements of each list. For example,

\[
\text{APPEND '(A B) '(C D)} \Rightarrow (A B C D) \\
\text{APPEND '(A) '(B (C))} \Rightarrow (A B (C)) \\
\text{APPEND '(A B) NIL) \Rightarrow (A B)
\]

LIST takes one or more arguments and makes a list with them as elements. Thus

\[
\text{LIST 'A 'B 'C 'D) \Rightarrow (A B C D)} \\
\text{LIST '(A B) '(C D)) \Rightarrow ((A B) (C D)) \\
\text{LIST '(A B) NIL) \Rightarrow ((A B) NIL)
\]

1.4 Tree Structures

We now explore several common cases involving the decomposition and rebuilding of LISP data structures.

Consider the class of LISP arithmetic expressions:

1. A number is an arithmetic expression.
2. If \( exp1 \) and \( exp2 \) are arithmetic expressions, then \( + \ exp1 \ exp2 \) is an arithmetic expression.
3. If \( exp1 \) and \( exp2 \) are arithmetic expressions, then \( * \ exp1 \ exp2 \) is an arithmetic expression.

Now suppose we wanted a function that would compute the value of an arbitrary arithmetic expression. We define a new LISP function COMPUTE to do this. It takes a LISP arithmetic expression and computes its value. Obviously, COMPUTE is imitating part of the behavior of EVAL.

A definition of COMPUTE is given in Figure 1.5.

The LISP function NUMBERP returns true if its argument is a number, and false otherwise.

The reserved word DEFUN is used to define new functions. In this case, we are defining a function named COMPUTE that takes one argument. The symbol AE will be bound to that argument when the function is called. The rest of the definition is an expression giving the value of COMPUTE. This expression is a COND-form.

In general, DEFUN-expressions are written like this:

\[
\text{(DEFUN name \{symbol\} \{expression\})}
\]

This defines name as a function with the symbols as the names of the function’s local variables (also called formal variables), and a function body
(DEFUN COMPUTE (AE)
  (COND ((NUMBERP AE) AE)
        ((EQUAL (CAR AE) '+)
         (+ (COMPUTE (CAR (CDR AE)))
            (COMPUTE (CAR (CDR (CDR AE))))))
        ((EQUAL (CAR AE) '*)
         (* (COMPUTE (CAR (CDR AE)))
            (COMPUTE (CAR (CDR (CDR AE))))))
        ))

FIGURE 1.5 Function to evaluate Lisp arithmetic expressions

made up of the expressions. The result of calling name with a set of arguments is found by binding the values of the arguments to the local variables, evaluating the expressions of the function body in order, and returning the value of the last expression evaluated.

In the definition of COMPUTE, notice that each clause of the COND exactly mirrors each clause of the recursive definition of "arithmetic expression." The first clause tests whether AE is a number. If it is, then the value of the arithmetic expression is just that number.

The second clause checks the CAR of the expression to see whether it is the symbol +. If it is, then the value of the whole expression is the sum of the values computed for the two subexpressions: (CAR (CDR AE)), which is the second element of AE, and (CAR (CDR (CDR AE))), which is the third.

* is handled like +, except that the two subexpression values are multiplied.

At this point it becomes relevant to introduce two more abbreviations. It is somewhat tiresome to have to write (CAR (CDR (CDR . . . .))), so Lisp allows you to abbreviate such expressions by just writing (CADDR . . .). In general, CxxxxXR may be used, where the x’s are replaced with A’s and D’s. (In most versions of Lisp, at most four A’s and D’s may be written.) The most interesting part of COMPUTE is its use of recursion. Many people have trouble grasping recursion at first. It helps to realize that recursive procedures are most often applied to recursively defined data structures that are to be decomposed. If you can grasp the data-structure definition, the program to operate on it will be clear. (Consider trying to write COMPUTE without using recursion.)

This example is also a demonstration of how easy it is to write language interpreters in Lisp. In this case, of course, the language being defined is a subset of Lisp itself, but this need not always be the case. It is not uncommon in Lisp to have programs which operate on other Lisp programs, as we shall see.
1.5 Lists

We can formally define the notion of a *list* as follows:

1. **NIL** is a list.
2. If *expression* is an expression and *list* is a list, then the result of 
   \( \text{(CONS expression list)} \) is a list.

Since this is a recursive definition, it is natural to think of using recursion 
when doing operations on lists. For example, Figure 1.6 defines the function 
**SQUARE-LIST** which takes a list of numbers and returns a list of their 
squares.

```lisp
(defun square-list (l)
  (cond ((null l) nil)
        (t (cons (square (car l))
                  (square-list (cdr l))))))

(defun square (n) (* n n))
```

**FIGURE 1.6** Function to square a list of numbers

(*NULL expression*) is equivalent to (*EQUAL expression NIL*).

In **SQUARE-LIST**, the first element of the list *L* is squared and put on the 
front of the list obtained by applying **SQUARE-LIST** to the rest of *L*. For 
example, (**SQUARE-LIST '(2 3)**) leads to the following chain of 
evaluation:

\[
\begin{align*}
\text{(SQUARE-LIST '(2 3))} & \quad = \quad \text{(CONS (SQUARE 2) (SQUARE-LIST '(3))} \\
& \quad = \quad \text{(CONS 4 (CONS (SQUARE 3) (SQUARE-LIST NIL))} \\
& \quad = \quad \text{(CONS 4 (CONS 9 NIL))} \\
& \Rightarrow \quad (4 \ 9)
\end{align*}
\]

In the definition of **SQUARE-LIST**, there is one **COND**-clause for each 
clause of the recursive definition of lists. If the list *L* is empty then the list of its 
squares is empty, too. Otherwise, the list of squares is obtained by **CONS**ing the 
square of the first element onto the squares of the rest of the elements.

The function **SQUARE-LIST** takes a list and transforms it into a list of 
equal length. It is just as easy to write a function that selects some elements and 
ignore others, thereby returning a list that is longer or shorter than the input list.
For example, Figure 1.7 defines \texttt{SQRT-LIST}, which takes a list of positive and negative numbers and returns a list of the square roots of the non-negative elements of the list.

\begin{verbatim}
(DEFUN SQRT-LIST (L)
  (COND ((NULL L) NIL)
    ((< (CAR L) 0) (SQRT-LIST (CDR L)))
    ((EQUAL (CAR L) 0)
      (CONS 0 (SQRT-LIST (CDR L))))
    (T (CONS (SQRT (CAR L))
       (CONS (- (SQRT (CAR L)))
            (SQRT-LIST (CDR L))))))

FIGURE 1.7 Function to take the square-root of each of a list of numbers
\end{verbatim}

This function has one test for the empty list, and three \texttt{COND}-clauses for the recursive part of the list definition. If the \texttt{CAR} of \texttt{L} is less than zero, we skip over it by just taking \texttt{SQRT-LIST} of the \texttt{CDR}. If it’s zero exactly, we \texttt{CONS} a zero onto the remaining square roots. If it’s positive, we have to include both the positive and negative roots. Thus one input list element can lead to zero, one, or two output elements. For example,

\[(\texttt{SQRT-LIST '(-9 0 4 -25)}) \Rightarrow (0 2 -2)\]

As an example of another list-building function, we shall define the function \texttt{REVERSE}. \texttt{(REVERSE list)} returns a list of the elements of \texttt{list} backwards. It already exists in LISP but it is instructive to define it ourselves.

One way of defining it is given in Figure 1.8.

\begin{verbatim}
(DEFUN REVERSE (L)
  (LET ((RESULT NIL))
    (LOOP (IF (NULL L) (RETURN RESULT))
      (SETQ RESULT (CONS (CAR L) RESULT))
      (SETQ L (CDR L))))

FIGURE 1.8 Function to reverse a list
\end{verbatim}

Notice how in an iterative loop, the function \texttt{CONS} naturally builds a list backwards.
Notice also that REVERSE binds values to symbols in three different ways. First, L is the formal variable of the function and is bound to whatever list REVERSE is applied to. Second, RESULT is the local variable of the LET and is bound to NIL initially. Third, during the loop both RESULT and L are explicitly assigned new values with the reserved word SETQ.

Both L and RESULT are local to the function REVERSE. That is, they are unaffected by the values of L and RESULT in any other function or expression, nor do they affect such values. It is a matter of good programming practice to make sure that all the variables used in a function are local, if at all possible, in order to avoid unexpected interferences between functions.

1.6 Mapping Functions

The control structure of SQUARE-LIST is so common that LISP provides facilities for doing it more directly. We do this with the function MAPCAR.

\[
\begin{align*}
(\text{DEFUN}\ SQUARE-LIST\ (&)\\
\quad (\text{MAPCAR}\ (\text{FUNCTION}\ SQUARE\ L))
\end{align*}
\]

FUNCTION is a reserved word. It's very similar to QUOTE, but we use it whenever we specifically want to refer to a LISP function, as opposed to a symbol or other S-expression. So far, we've seen function-names used in function definitions, i.e., DEFUN-forms, and in function calls, where the function-name is the CAR of a list that's being evaluated. But this is the first time we've seen a function-name passed as an argument.

Since FUNCTION is so common, LISP uses a special notation for it. An expression written as # \text{'exp} is transformed by the function READ into (FUNCTION \text{exp}). So we could rewrite the definition of SQUARE-LIST this way:

\[
(\text{DEFUN}\ SQUARE-LIST\ (&)\\
\quad (\text{MAPCAR}\ #'\text{SQUARE~}\ L))
\]

MAPCAR is a mapping function. In its simplest form, it takes two arguments, a function and a list, and returns a new list consisting of the values of the function applied to the elements of the original list.

\[
(\text{MAPCAR}\ #'\text{SQUARE}\ '(3\ -2\ 12\ 4))
\]

\[
=\ \text{(LIST}\ (\text{SQUARE}\ 3)\\
\quad (\text{SQUARE}\ -2)\\
\quad (\text{SQUARE}\ 12)\\
\quad (\text{SQUARE}\ 4))
\]

\[
\Rightarrow (9\ 4\ 144\ 16)
\]

MAPCAR could be defined as in Figure 1.9.
(DEFUN MAPCAR (F L)
  (COND ((NULL L) NIL)
        (T (CONS (FUNCALL F (CAR L))
             (MAPCAR F (CDR L))))))

FIGURE 1.9 Definition of MAPCAR (one list)

FUNCALL is a function that takes an arbitrary number of arguments. The first one should evaluate to a function. FUNCALL calls that function, passing to it the values of the rest of FUNCALL's arguments.

For example

(FUNCALL (CAR (LIST #'CONS #'APPEND #'FOO))
     'A NIL)

⇒ (CONS 'A NIL)

⇒ (A)

In general

(FUNCALL function {expression}*)

⇒ (value-of-function {expression}*)

There are other mapping functions: MAPC, MAPCAN, EVERY, and SOME. In COMMON LISP, MAPCAR, MAPCAN, and MAPC actually can take more than two arguments. The first one must be a function. The rest are all lists. If that function takes 2 arguments, for example, then you would call MAPCAR with the name of the function and 2 lists. For example,

(MAPCAR #'CONS '(A B C) '(X Y Z))

⇒ (LIST (CONS 'A 'X)
         (CONS 'B 'Y)
         (CONS 'C 'Z))

⇒ ((A X) (B Y) (C Z))

1.7 LAMBDA Expressions

Suppose we want to take a list of numbers and produce a new list of the same length, consisting of the symbols POS, NEG, or ZERO, such that POS appears in the positions corresponding to positive numbers, NEG in those for negative numbers, and ZERO for the zeros. The code in Figure 1.10 would do this.

However, the function SYM-SIGN will probably never be used anywhere else. It would be more reasonable if we could write the code for it directly in the
MAPCAR expression, without having to make up a name. In fact, we can do this, as shown in Figure 1.11.

```
(MAPCAR #'(LAMBDA (N)
    (COND ((< N 0) 'NEG)
         ((EQUAL N 0) 'ZERO)
         ((> N 0) 'POS)))
    L)
```

FIGURE 1.11 Example of mapping a LAMBDA-expression

The general construct, called a functional constant or LAMBDA-expression is

```
(LAMBDA (symbol*) expression*)
```

It is a function without a name. When called, it is evaluated just like a function, by first binding the symbols (i.e., the local variables) to the arguments, and then evaluating the expressions in the body of the LAMBDA.

This may seem strange at first, because in most programming languages, all functions have to have names. In LISP, however, a function can be created and used without ever giving it a name, in much the same way as expressions can. Consider the expression

```
(SETQ X (+ (* 5 3) 2))
```

We could get the same result by writing

```
(SETQ TEMP (* 5 3))
(SETQ X (+ TEMP 2))
```

That is, we might save the value of (* 5 3) by assigning it to a variable, but we don't have to, and unless we're going to need that value in several places, we probably don't want to. Instead, we would use the expression (* 5 3).
directly as part of another expression. The same thing is true for functions in LISP. While we usually want to save function definitions by giving them names, we don't always. We can use them directly, wherever a function would be used.

For example, we could write

```
(FUNCALL #'(LAMBDA (x) (+ x (/ 1 x)))
       (- (SQUARE A) (SQUARE B)))
```

which is equivalent to

```
(+ (- (SQUARE A) (SQUARE B))
    (/ 1 (- (SQUARE A) (SQUARE B))))
```

except that we avoid recomputing $A^2$ minus $B^2$ by assigning it to the local variable `x`. Using local variables to avoid extra calculations is a handy technique which will be discussed later on.

This use of "nameless" functions is very important in LISP, and a special notation is allowed for it.

```
((LAMBDA (x) (+ x (/ 1 x)))
  (- (SQUARE A) (SQUARE B)))
```

We can eliminate the call to FUNCALL, and we don't have to quote the LAMBDA-expression. This is, in fact, exactly what LET does. We could rewrite this expression as

```
(LET ((x (- (SQUARE A) (SQUARE B)))
       (+ x (/ 1 x)))
```

Finally, DEFUN is just a way of attaching a globally accessible name to a LAMBDA-expression. That is, the following two expressions have approximately the same effect:

```
(DEFUN SQUARE (N) (* N N))
(SETF (SYMBOL-FUNCTION 'SQUARE)
       #'(LAMBDA (N) (* N N)))
```

In some LISPs, including COMMON LISP, the function associated with a symbol is stored separately from other information about that symbol (such as its value). In COMMON LISP, SYMBOL-FUNCTION simply names the place where the function is stored. SETF is a generalization of SETQ, allowing us to store values in "places." We'll see more of SETF shortly.

1.8 Atoms

Numbers and symbols are "primitive": for most purposes, they have no internal structure, and they each have their own particular set of basic operators.
For numbers, there are the various arithmetic functions such as + and *, and the ordering functions such as > and <. For symbols, there are functions for attaching information to them. This is important because symbols have one very useful property — each symbol refers to one, unique data structure. Once information is attached to a symbol, you can get at it easily by giving the symbol’s name. For example, once you have attached a function definition to the symbol SQUARE-LIST, you can access and use that definition whenever you want, just by using the name SQUARE-LIST.

1.9 Inside LISP

Until now we have avoided saying anything about how LISP actually represents lists inside the machine. To some extent we never have to worry about how LISP really does it, but we do need to describe an abstract data structure that is actually quite close to the real thing.

This abstract data structure is called a dotted pair, or sometimes a CONS-cell. A dotted pair is a pair of pointers. A pointer is either the null pointer (i.e., it is a pointer to NIL) or it is a “link” to another dotted pair or an atom. The first pointer is called the CAR of the pair and the second is called the CDR.

There are two ways commonly used to write a dotted pair. First, there is a graphical notation. Graphically, a dotted pair is written as a rectangle divided into a left and a right box. The null pointer is written as a diagonal slash in the appropriate box of the rectangle. An atom is written by putting its printed representation (e.g., a symbol’s name, a number’s value) in the appropriate box. A pointer is written by drawing an arrow from the appropriate box to the next dotted pair. Figure 1.12 shows a dotted pair whose CAR points to a dotted pair of the symbols A and B, and whose CDR is the null pointer.

The LISP notation for dotted pairs is to write a left parenthesis, followed by the LISP notation for the data structure pointed to by the CAR, followed by a period, followed by the LISP notation for the data structure pointed to by the CDR, followed by a right parenthesis. The null pointer is written as NIL, and atoms are written with their names.

Thus the graphical structure in figure 1.12 is written as

\[(\text{A} \cdot \text{B}) \cdot \text{NIL}\]

A list is a special case of dotted-pair structure. A list is either

1. the null pointer, or
2. a dotted pair whose CAR is an expression and whose CDR is a list.

Compare this data structure definition with the CONS procedure one given in the section on “Lists.”
Thus the following are lists:

- NIL
- (A, NIL)
- (NIL, NIL)
- (A, (B, NIL))
- ((A, NIL), NIL)
- ((A, NIL), (B, NIL))

The following are not:

- A
- (A, B)
- (NIL, A)
- (A, (B, C))
- ((A, NIL), B)
- ((A, B), (C, D))

Even though they are more restricted, lists are much more commonly used in LISP than dotted pairs. Almost anything that can be represented with dotted pairs can be represented with lists, although some extra space may be wasted to hold the extra NILs. Many functions, such as the mapping functions, require that their arguments be lists. For example, the following expression is erroneous because the second argument of the MAPCAR is not a list.

```
(MAPCAR #'PRINT '(A, B))
```

Since lists are commonly used but look ugly in dotted-pair notation, LISP uses a different system, called list notation, which includes dotted-pair notation as a special case. It is the notation we have been using all along in this chapter.

The algorithm for writing a (noncircular) dotted-pair structure in list notation is:
1. Set a pointer Q to the beginning of the dotted-pair structure and write a left parenthesis.

2. Write the list notation for the data structure pointed to by the CAR of Q, and reset Q to the CDR of Q.

3. If Q is now the null pointer, then write a right parenthesis. If Q is an atom (other than NIL), then write a space, a period, a space, Q's name, and a right parenthesis. If Q is not an atom, write a space and go to step 2.

List notation in general is a lot easier to read than the equivalent dotted pair notation.

\[
\begin{align*}
(A \cdot \text{NIL}) & = (A) \\
(A \cdot B) & = (A \cdot B) \\
(\text{NIL} \cdot \text{NIL}) & = (\text{NIL}) \\
(A \cdot (B \cdot \text{NIL}))) & = (A \cdot B) \\
((A \cdot \text{NIL}) \cdot (B \cdot \text{NIL})) & = ((A) \cdot B) \\
(A \cdot (B \cdot C)) & = (A \cdot B \cdot C)
\end{align*}
\]

Exercise 1.1
Define the function DOT-PRINT, which will take an arbitrary S-expression and print it in dot notation. Use these expressions:

- `(PRINC "(")` to print a left parenthesis,
- `(PRINC ")")` to print a right parenthesis,
- `(PRINC ",")` to print a period,
- `(PRINC "\)")` to print a space,
- `(PRINC atom)` to print `atom`.

These functions will be described later in the chapter on input/output in LISP.

Exercise 1.2
Define the function LIST-PRINT, which will take an S-expression and print it in standard LISP list notation. Use the same printing expressions as given for DOT-PRINT. Suggestion: write two functions: LIST-PRINT, which starts at step 1; and LIST-PRINT-2, which starts at step 2. This will simplify implementing step 3.

A symbol names a location where a value can be stored, and we can use `SETQ` to store a value in that location. A CONS-cell is a pair of locations, too; to store values there, we use `SETF`. For example,
(SETQ X (CONS 'A 'B)) \rightarrow (A , B)
(SETF (CAR X) 'C) \rightarrow C
X \rightarrow (C , B)
(SETF (CDR X) 'D) \rightarrow D
X \rightarrow (C , D)

SETF, unlike the other functions we have described in this chapter, actually changes internal pointers. This means that its effects are global and permanent. It should therefore be used with care.

The function CONS in Lisp could be defined in terms of SETF if we had available a function that would return an unused dotted-pair cell whenever we wanted it. Suppose this function (which doesn't really exist) were called NEW-CELL. Then a definition of CONS would be

(DEFUN CONS (X Y)
  (LET ((Z (NEW-CELL)))
    (SETF (CAR Z) X)
    (SETF (CDR Z) Y)
    Z))

That is, we would get a new cell, replace the CAR of it with X, and replace the CDR of it with Y. The fact that CONS gets a new cell rather than changing existing cells can be important.

For example, suppose we defined the functions INSERT-1 and INSERT-2, both taking an expression and a dotted pair and returning a dotted pair of the expression and the CDR of the original dotted pair. We define these functions as

(DEFUN INSERT-1 (X L) (CONS X (CDR L)))
(DEFUN INSERT-2 (X L) (SETF (CAR L) X) L)

INSERT-1 builds a new dotted pair, while INSERT-2 changes the old one.

With these definitions, the following sequence of evaluations would occur.

(SETQ X (CONS 'B 'C)) \rightarrow (B , C)
(INsert-1 'A X) \rightarrow (A , C)
X \rightarrow (B , C)
(INsert-2 'A X) \rightarrow (A , C)
X \rightarrow (A , C)

Notice that INSERT-2 has changed the value of X! Furthermore, if we continued this sequence with

(INsert-2 X X)

we would create a circular list. The CAR of X would be X. Graphically this X would be set to the structure in Figure 1.13.
Exercise 1.3
(Very difficult) Write the circular list \( X \) in LISP dotted-pair notation.

One other useful function that changes the internal LISP structures is NCONC, which is like APPEND in that it concatenates two lists together. However, NCONC physically alters the first list so that its final CDR, which used to point to NIL, now points to the second list.

Exercise 1.4
Define NCONC using SETF and CDR. If we set \( X \) to \( (\text{LIST} \ \text{NIL}) \), what is the effect of \( \text{NCONC} \ X \ X \)?

You should never use SETF or any other "destructive" operator to change any part of a constant, such as a quoted list or a literal string. For example, it would seem that \( \text{SMALLP} \ 2 \) should always be true, given the following definition:

\[
(\text{DEFUN SMALLP} \ (N) \\
\quad (\text{MEMBER} \ N \ '(0 \ 1 \ 2 \ 3 \ 4 \ 5)))
\]

But consider the effect of calling \( \text{PERVERSE} \ 2 \):

\[
(\text{DEFUN PERVERSE} \ (N) \\
\quad (\text{LET} \ ((X \ \text{SMALLP} \ N)) \\
\quad \quad (\text{IF} \ X \ (\text{SETF} \ (\text{CAR} \ X) \ \text{NIL})))
\]

After \( \text{PERVERSE} \ 2 \) is called, \( \text{SMALLP} \ 2 \) will return NIL! In fact, if your LISP system can display the function definition for SMALLP, you might see this after PERVERSE was called:

\[
(\text{DEFUN SMALLP} \ (N) \\
\quad (\text{MEMBER} \ N \ '(0 \ 1 \ NIL \ 3 \ 4 \ 5)))
\]

If you didn't know what was going on, you might start believing in ghosts. This short example is unrealistic, but the same problem can occur in large
systems, and can be difficult to track down. It's safest to be conservative and imagine that constants will be stored in read-only memory, and that you will get some drastic hardware fault if you attempt to change them.

1.10 Equality

In the last section we noted that CONS uses fresh cells when it builds lists. That means that it is possible to build two lists that look the same but are actually separate data structures.

For example, this code

```
(SETQ L1 (CONS 'A NIL)) ⇒ (A)
(SETQ L2 (CONS 'A NIL)) ⇒ (A)
(SETQ L3 L1) ⇒ (A)
```

will generate two lists, L1 and L2, with the same shape and the same components. Since each was built with a separate call to CONS, L1 and L2 will each occupy a different CONS cell. However there was no call to CONS when L3 was assigned the value of L1. Hence L1 and L3 will have the same CONS cell. Should LISP consider L1 and L2 to be equal or not? They are clearly not "as equal" as L1 and L3.

To resolve this dilemma, COMMON LISP provides a hierarchy of predicates that successively broaden the definition of equality. The most restrictive one is EQ:

```
(EQ exp1 exp2)
```

EQ returns T if exp1 and exp2 are exactly the same object. If two symbols have the same name, for example, then they are exactly the same object, so EQ is the test to use when comparing symbols. In the case of CONS-cells, EQ returns T if the CONS-cells are, in fact, the same CONS-cell, not merely CONS-cells with the same components. Hence

```
(EQ L1 L2) ⇒ NIL
(EQ L1 L3) ⇒ T
```

The next equality predicate in the hierarchy is EQL:

```
(EQL exp1 exp2)
```

EQL returns T if exp1 and exp2 are EQ, or if they are numbers with the same type and value, or if they are the same characters.

Next, there's EQUAL:

```
(EQUAL exp1 exp2)
```

EQUAL returns T if exp1 and exp2 are EQL, or if they're CONS-cells whose components (i.e., CAR and CDR) are themselves EQUAL, or if they're
character strings with the same length and same characters. \(\text{EQUAL}\) is also used to compare some other data structures, such as arrays, that we haven’t seen yet.

Note what happens now when we compare our lists with \text{EQUAL}:

\[
\begin{align*}
\text{(EQUAL L1 L2)} & \Rightarrow T \\
\text{(EQUAL L1 L3)} & \Rightarrow T
\end{align*}
\]

We can sketch the definitions of \text{EQL} and \text{EQUAL} in Lisp as shown in Figure 1.14.

\begin{verbatim}
(DEFUN EQL (X Y)
  (COND ((EQ X Y) ; Are they already EQ?
    T)                ; If so, then they’re EQL, too.
    ((AND (INTEGERP X) (INTEGERP Y)) ; Are they both integers?
      (= X Y)) ; Then compare integers.
    (...) ; (Similar checks for other types of numbers)
    ((AND (CHARACTERP X) (CHARACTERP Y)) ; Are they both characters?
      (CHAR= X Y)) ; Then compare characters.
    (T NIL))) ; Otherwise, they’re not EQL.

(DEFUN EQUAL (X Y)
  (COND ((EQL X Y) ; Are they already EQL?
    T)            ; If so, then they’re EQUAL, too.
    ((AND (CONSP X) (CONSP Y)) ; Are they both CONS-cells?
      (AND (EQUAL (CAR X) (CAR Y)) ; Then compare their CARs...
        (EQUAL (CDR X) (CDR Y)))) ; ... and their CDRs.
    ((AND (STRING X) (STRING Y)) ; Are they both strings?
      (STRING= X Y)) ; Then compare strings.
    (...) ; (A few other tests follow here.)
\end{verbatim}

FIGURE 1.14 Sketches of definitions of EQL and EQUAL
The least restrictive equality predicate is called `EQUALP`. Two objects are `EQUALP` if they are `EQUAL`, or if they are the same character (regardless of case), or if they are numbers with the same value (regardless of type), or if they data structures of the same type whose components are themselves `EQUALP`. (We'll see an example of this in Chapter 3.)

As you can see, the question of equality becomes complicated when there are so many kinds of data types in the language. In some of the earliest Lisps, `(EQ X Y)` was true when `X` and `Y` were stored at the same machine address, and integers that could be represented in less than 16 bits, for example, were implemented as addresses, so they were `EQ`, too. Other comparisons were left up to `EQUAL`, and Lisps differed in the way they compared the integer 0, for example, with the floating-point number 0.0, or upper-case letters with lower-case letters.

More recent Lisps tend to have a variety of type-specific equality predicates, such as a test for comparing two integers. "Generic" equality predicates such as `EQUAL` are defined in terms of these more specific predicates as Figure 1.14 shows. From an efficiency standpoint, it is always better to use a type-specific predicate when you know in advance what the types are. One example in COMMON LISP is `=`, which is used for comparing two numbers. Other examples we will use include `STRING=` (for strings) and `CHAR=` (for characters). Up to now, we have used `EQUAL` for all our Lisp examples, but from now on, we will use type-specific predicates.

The main point of looking at `EQUAL`, however, was to point out the parallel between recursive data types (i.e., a data structure whose components are of the same data type, such as a CONS-cell whose CAR and CDR are CONS-cells) and recursive programs (`EQUAL` calls itself).

### 1.11 Local versus free variables

Consider the code in Figure 1.15. The function `PRINT-WARNING` takes an expression, prints some warning message with the expression, reads a "response" (presumably something you type at the terminal), and returns either the expression that you typed if your response is non-NIL, or else the original expression.

In this function, the variables `BAD-FORM` and `RESPONSE` are local, lexically scoped variables. `BAD-FORM` is a LAMBDA-variable, since functions are really LAMBDA-expressions; it is bound to the value that is passed to `PRINT-WARNING`. `RESPONSE` is also a LAMBDA-variable, since `LET` is equivalent to a LAMBDA-expression; it is initially bound to NIL when the `LET`-form is entered, and then set to whatever you type, i.e., whatever `READ` returns. The binding between these names and values exists only inside the function `PRINT-WARNING`. (To be more precise, it is the binding of values
to names that is lexical.) Other functions might also have local variables named BAD-FORM and RESPONSE, but any bindings those functions might establish are completely separate from these bindings, even if those functions are called while PRINT-WARNING is being executed.

The variable *WARNING-MSG*, on the other hand, is a non-local or free variable in PRINT-WARNING. That is, it is not a locally bound variable in any LET or LAMBDA in PRINT-WARNING. The form

```
(DEFVAR *WARNING-MSG* "Problem:"
)
```

declares *WARNING-MSG* to be a special variable and assigns it an initial, global value, the string "Problem:". If we called PRINT-WARNING from the top level, this is the string that would be printed. But special variables can be temporarily bound to other values, and the temporary binding is accessible everywhere. This is what happens in the function ATOM-WARNING. *

*WARNING-MSG* is bound to the string "Unrecognized atom:" by the LET-form, and this binding remains in effect while the code inside the LET-form is executed. That code consists of a call to PRINT-WARNING, so if PRINT-WARNING is called from ATOM-WARNING, the string "Unrecognized atom:" will be printed. As soon as we return from executing the LET-form inside ATOM-WARNING, however, the previous value of *WARNING-MSG* is restored. This is called dynamic scoping; again, it is the binding that is dynamic.

It is necessary to declare *WARNING-MSG* to be special. If the DEFVAR had been omitted, then PRINT-WARNING would cause an error when it tried to print the value of *WARNING-MSG*, since *WARNING-MSG* would be "unbound." In ATOM-WARNING, *WARNING-MSG* would be a local,
lexically bound variable whose value is never used. The fact that PRINT-WARNING is called while this lexical binding is in effect makes no difference; the lexical binding is invisible outside ATOM-WARNING, so PRINT-WARNING would still get an "unbound variable" error.

Thus, while we normally think of LET and LAMBDA as creating lexical bindings for local variables, we see now that they may not; they are also used to create dynamic bindings for special variables. This is why special variables must be declared; the ambiguity must be resolved, since the mechanisms for creating lexical and dynamic bindings are different.

In some Lisps, this ambiguity doesn’t arise; they rely on different syntax for lexical and dynamic bindings. In Chapter 2, we’ll see how to define a macro that works like LET but automatically declares its variables to be special.

However, there is a real question as to whether one wants to make it easy to use special variables. We believe that they should be avoided. They incur substantially more overhead than lexical variables, but more importantly, special variables tend to make for bad code.

For one thing, they make code harder to debug. If a variable is local, we know that its value cannot change between one use and the next inside a function, unless that function changes it. With special variables, this is not true. If we call a subfunction it may, either accidentally or deliberately, change the value of a special variable.

Special variables also make code harder to read since you cannot ascertain the significance of a special variable by looking at the immediately surrounding code. Special variables should therefore always be given mnemonic, visually striking names (such as prefix and postfix asterisks), and they should be set apart in the code. Declaring them special at the start of your code is a good way of doing the latter.

1.12 LISP style

The functions defined in this book are written according to a number of general stylistic rules.

Indentation

Unindented LISP code is unreadable by anyone except the machine. Misleadingly indented code is even worse. This book contains lots of explicit examples of indentation. The general rule is to use spaces liberally — it takes LISP almost no time at all to ignore them. Lines should be short, with only one expression, unless there are several atoms in a row.

When you have to break an expression up into several lines, there are several standard options for where to start the second line. Figure 1.16 has two alternative indentations of the definition of SQUARE-LIST.

There are some special rules for indentation. Function definitions should
always have the DEFUN, the function name, and the variable list on the same line. If the rest of the definition is too long, it is put on succeeding lines, indented at least one space.

Naming

Choose variable and function names so that they suggest something (the right thing!) to the naive reader. Don’t call a function to embed a graph in a plane FEGP — call it GRAPH-EMBED. But don’t expect names to replace comments. Good names can only remind the reader of what is happening.

Set up some conventions for naming things consistently. For example, have some standard way of naming global variables. In this book, we use the convention that all global and special variables have asterisks ("*")) as the first and last characters.

Boolean variable and functions (i.e., whose values are either true or false) should be given names like F00P or F00-P (P for predicate) or IS-F00 or F00?. COMMON LISP uses the letter P as a suffix for almost all predicates. Sometimes a hyphen precedes the P, depending on whether the predicate already contains a hyphen, whether it is a type-specific predicate, and whether it is generated by DEFSTRUCT; see the COMMON LISP manual [102] for examples.

Comments

In COMMON LISP, comments are signalled by a semicolon. When READ sees a semicolon, it ignores all the remaining characters on that line. Thus you can write anything you want on a line after a semicolon.

Programs in Lisp require as much commenting as in any other language. Try to have a structure to your comments, such as specifying in a fixed order, for each function, what its input variables represent, what kinds of values it is expected they will have, what global variables the function uses, and what kind of value the function returns.
Be sure to keep your comments up to date. If you are changing your code a lot, remove the comments until things settle down. A misleading comment can be worse than no comment at all.

Function Size

Functions should be small, certainly never more than twenty lines. If things start to nest deeply, e.g., if you have ten levels of parentheses, then start breaking up the subexpressions into subfunctions, in order to give the poor reader a rest.

Beware of indiscriminate break-up however. If you have to use several global variables, or more than three or four arguments to the subfunctions, or have trouble coming up with better names than F001, F002, and so on, then you are probably breaking things up the wrong way. Try using FLET and LABELS to define local functions, just as you would use LET to define local variables.

Cleverness

Avoid it. Clever tricks are those that depend on how a particular LISP system is implemented. For example, your LISP may implement atoms as lists, but not all Lisps do this, and hence any code using this fact will not be transportable. As another example, in UCI LISP, MAPC always returns NIL, but in COMMON LISP it returns the value of the second expression evaluated. Thus, in UCI LISP the following code would call MAPC and then BAZ, but in COMMON LISP it would call MAPC and then F00:

```lisp
(COND ((MAPC (FUNCTION TEST) '(A B C))
       (F00))
      (T (BAZ)))
```

LISP has several functions that, like MAPC and PRINT, exist solely for their side-effects. It's not a good idea to depend on their values at all.

To paraphrase Samuel Johnson, we advise you to look over your code very carefully, and whenever you find a part that you think is particularly fine, strike it out.

1.13 Keywords

In the Lisp programs we've seen so far, functions are called with the same number of arguments as their LAMBDA-definitions, and in the same order. So when we write

```lisp
(DEFUN LIMIT (N LOWER-BOUND UPPER-BOUND)
             (MAX LOWER-BOUND (MIN N UPPER-BOUND)))
```
then we call it like this:

```lisp
(LIMIT -4 1 10) ⇒ 1
```
Inside the body of LIMIT, the variable N would have the value -4, LOWER-BOUND would have the value 1, and UPPER-BOUND would have the value 10. COMMON LISP allows us to be more flexible, however, and this flexibility is useful in a number of ways.

We could define LIMIT this way:

```
(DEFUN LIMIT (N &KEY LOWER-BOUND UPPER-BOUND)
  (MAX LOWER-BOUND (MIN N UPPER-BOUND)))
```

and call it this way:

```
(LIMIT -4 :LOWER-BOUND 1 :UPPER-BOUND 10) => 1
```

to achieve the same effect as before. But we could also switch the order of the keywords, like this:

```
(LIMIT -4 :UPPER-BOUND 10 :LOWER-BOUND 1) => 1
```

and get the same result.

&KEY is a lambda-list keyword. There are several others, such as &OPTIONAL and &REST, all defined in the COMMON LISP manual. We discuss some of them at the end of this chapter.

:LOWER-BOUND and :UPPER-BOUND are called keywords. As far as LISP is concerned, keywords are self-evaluating constants, like numbers; the value of the symbol :LOWER-BOUND is :LOWER-BOUND itself. Here, they serve to name the variables LOWER-BOUND and UPPER-BOUND, which are called the keyword parameters.

1 and 10 are the keyword values, that is, the values associated with the keyword parameters.

This convention enables us to specify the arguments by naming their positions, instead of having to remember the order in which to pass the arguments.

Keyword values are optional. If you don’t supply them, they default to NIL (or a value that you specify). Thus,

```
(LIMIT -4 :UPPER-BOUND 10)
```

would be equivalent to

```
(LIMIT -4 :LOWER-BOUND NIL :UPPER-BOUND 10)
```

Using NIL as a default in this example would cause an error when MAX was called. However, you can specify what the default values should be.

```
(DEFUN LIMIT (N &KEY (LOWER-BOUND 0)
  (UPPER-BOUND 100))
  (MAX LOWER-BOUND (MIN N UPPER-BOUND)))
```

In this example, if you don’t specify a value for LOWER-BOUND, it will have the value 0. If you don’t specify a value for UPPER-BOUND, it will have the value 100.
Note that the first parameter, \( N \), is \textit{required}. You can’t specify its value by calling
\[
\text{LIMIT :N 10 ...}
\]
In all the examples prior to this section, all the parameters were required. Of course, you can make \textit{all} the parameters into keyword parameters by writing
\[
\text{DEFUN LIMIT (&KEY N LOWER-BOUND UPPER-BOUND) ...}
\]
That is, the symbol \&KEY separates the required parameters from the keyword parameters.

An example of a useful LISP function that uses keywords is \texttt{MEMBER}.

\[
\text{(MEMBER item list)}
\]
\texttt{MEMBER} tests to see whether \textit{item} is in \textit{list}. If not, \texttt{MEMBER} returns \texttt{NIL}. If so, \texttt{MEMBER} returns the tail of the list, starting with \textit{item}. Thus,

\[
\begin{align*}
\text{(MEMBER 1 '(2 3 4))} & \Rightarrow \text{NIL} \\
\text{(MEMBER 3 '(2 3 4))} & \Rightarrow (3 4)
\end{align*}
\]

But what exactly does it mean for an item to be \textit{in} a list? Presumably, we want to test whether \textit{item} is “the same” as some element of \textit{list}, but as we have seen, there are various notions of what it means to be “the same.”

\texttt{MEMBER} has a keyword parameter called \texttt{TEST} that is used for comparing \textit{item} to each element of \textit{list}. The default value for \texttt{TEST} is the predicate \texttt{EQL}. We could use any other test to change the behavior.

Suppose we wanted to look through a list of numbers, \( L \), for one that was “close” to some other number, \( N \), where “\( x \) is close to \( y \)” meant that the difference between \( x \) and \( y \) was less than 3. We could write
\[
\begin{align*}
\text{(DEFUN PRETTY-CLOSE (X Y) (< (ABS (- X Y)) 3))} \\
\text{(MEMBER N L :TEST #'PRETTY-CLOSE)}
\end{align*}
\]
or in more compact form,
\[
\begin{align*}
\text{(MEMBER N L :TEST #'(LAMBDA (X Y) (< (ABS (- X Y)) 3)))}
\end{align*}
\]

Finally, keywords are also useful outside the context of lambda-lists. Since they are constants, they are useful as symbolic values:
\[
\begin{align*}
\text{(IF (EQ (NAME CHESS-PIECE) :ROOK) } \\
\text{MOVE-LATERALLY ... )} \\
\text{(SETF (FUEL-GAUGE) :FULL) } \\
\text{(IF (MEMBER COLOR '(' :BLUE :GREEN :RED) ... )}
\end{align*}
\]

For simple LISP programs, this has the same effect as using non-keyword symbols ("ROOK", "FULL", ...), but COMMON LISP has a mechanism called
a package that is especially useful for large programs. Symbols belong to packages, and the symbol \texttt{FULL} in package \textit{A} is \textit{not the same} as the symbol \texttt{FULL} in package \textit{B}. If a large program is written in different modules, each module can use a different package and thereby avoid collisions in names of functions. But when code from different modules \textit{want} to use the same symbolic constant, they can use keywords. Keywords are simply symbols that belong to the keyword package. \texttt{:FULL} is the symbol \texttt{FULL} in the keyword package, easily recognized by the leading colon. No matter what package you’re currently using, it’s always safe to use keywords. As Gertrude Stein would say, \\
\texttt{:ROSE EQ :ROSE EQ :ROSE}.

We also use keywords as \textit{syntactic markers}. \texttt{Lisp} allows you to extend the syntax of the language through the use of macros (see the next chapter), and keywords are an excellent means of separating the “noise words” from the rest of the expression. We’ll see an example of keywords in macros in Chapter 3.

\subsection*{1.14 More lambda-list keywords}

In the previous section, we saw the effect of using \&\texttt{KEY} in a lambda-list. Another useful lambda-list keyword is \&\texttt{REST}; the variable following \&\texttt{REST} will be bound to the list of remaining arguments. For example,

\begin{verbatim}
(DEFUN ALPHA (A B &REST Z) (LIST A B Z))
(ALPHA 1 2) \Rightarrow (1 2 NIL)
(ALPHA 1 2 3) \Rightarrow (1 2 (3))
(ALPHA 1 2 3 4) \Rightarrow (1 2 (3 4))

(DEFUN LIST (&REST ARG\texttt{S}) ARG\texttt{S})
; A possible definition for LIST
\end{verbatim}

You can define a function with \textit{optional} arguments by using the lambda-list keyword \&\texttt{OPTIONAL}. You can also specify what value should be used if the argument is omitted.

\begin{verbatim}
(DEFUN BETA (A &OPTIONAL B) (LIST A B))
(BETA 1) \Rightarrow (1 NIL)
(BETA 1 2) \Rightarrow (1 2)
(BETA) \Rightarrow error: A is not optional.
(BETA 1 2 3) \Rightarrow
    error: BETA takes at most two arguments.

(DEFUN GAMMA (A &OPTIONAL (B 13)) (LIST A B))
(GAMMA 1) \Rightarrow (1 13)
\end{verbatim}

There are other lambda-list keywords. The one that we’ll use most often is \&\texttt{BODY}. It’s just like \&\texttt{REST}, but it’s used with macros (see Chapter 2).
Bibliography

1. AAAI-84.  

2. Harold Abelson and Gerald Jay Sussman.  
   Structure and Interpretation of Computer Programs.  

3. Alfred V. Aho and Jeffrey D. Ullman.  
   Principles of Compiler Design.  

4. David Barstow.  
   Automatic construction of algorithms and data structures using a  
   knowledge base of programming rules.  

   An augmented state transition network analysis procedure.  
   In IJCAI-69 [45], page 557.

   New programming languages for artificial intelligence.  

7. Daniel G. Bobrow and Terry Winograd.  
   An overview of KRL, a knowledge representation language.  
8. R. S. Boyer and J S. Moore.
The sharing of structure in theorem-proving programs.
In Meltzer and Michie [70].

9. R. J. Brachman.
"I Lied about the Trees" or Defaults and Definitions in Knowledge Representation.

An Overview of the KL-ONE Knowledge Representation System.

A computational model of understanding.

12. Lee Brownston, Robert Farrell, Elaine Kant, and Nancy Martin.

Recursive Programming Techniques.
Reading, Massachusetts: Addison-Wesley, 1975.

Symbolic Logic and Mechanical Theorem Proving.

15. Eugene Charniak.
Toward a Model of Children's Story Comprehension.

A framed PAINTING: the representation of a common sense knowledge fragment.

17. Eugene Charniak and Drew V. McDermott.
Artificial Intelligence.

Artificial Intelligence Programming.

Visual Information Processing.
*Programming in PROLOG.*  

21. COGSCI-84.  

*Script Application: Computer Understanding of Newspaper Stories.*  

*Structured Programming.*  

24. Randall Davis.  
Applications of meta level knowledge to the construction, maintenance, and use of large knowledge bases.  

25. Randall Davis and Jonathan J. King.  
An overview of production systems.  

26. Thomas Dean.  
*An Approach to Reasoning about Time for Planning and Problem Solving.*  
New Haven: Yale Computer Science Department Research Report 433.

27. Johan de Kleer.  
Choices without backtracking.  
In AAAI-84 [1], pages 79-85.

AMORD: Explicit control of reasoning.  
*Proceedings of the Symposium on AI and Programming Languages.*  

29. Johan de Kleer, Jon Doyle, Guy L. Steele, Jr., and Gerald Jay Sussman.  
AMORD, a deductive procedure system.  

30. Johan de Kleer, Jon Doyle, Guy L. Steele, Jr., and Gerald Jay Sussman.  
AMORD, a deductive procedure system.  

The structure of the "THE" multiprogramming system.  
32. Edsger W. Dijkstra.  
The humble programmer.  

33. Jon Doyle.  
Truth maintenance systems for problem solving.  

34. G. W. Ernst and Allen Newell.  

STRIPS: a new approach to the application of theorem proving to problem solving.  
In IJCAI-71 [46], page 608.

36. Nicholas V. Findler, editor.  

37. Charles L. Forgy.  
Rete: A fast algorithm for the many pattern/many object pattern match problem.  

Conceptual generation.  
In Schank [93], page 289.

39. Ira P. Goldstein and R. Bruce Roberts.  
NUDGE: a knowledge-based scheduling program.  
In IJCAI-77 [48], pages 257-263.


A formal basis for the heuristic determination of minimum cost paths.  

42. Frederick Hayes-Roth.  
The role of partial and best matches in knowledge systems.  
In Waterman and Hayes-Roth [112], page 557.

43. Peter Henderson.  

44. Carl Hewitt.  
Procedural embedding of knowledge in PLANNER.  
In IJCAI-71 [46], page 167.
45. IJCAI-69.

46. IJCAI-71.

47. IJCAI-73.

48. IJCAI-77.
Proceedings of the Fifth International Joint Conference on Artificial Intelligence, MIT, 1977.

49. IJCAI-79.
Proceedings of the Sixth International Joint Conference on Artificial Intelligence, Tokyo, 1979.

50. Donald E. Knuth.

51. Donald E. Knuth.

52. Robert Kowalski.
Logic for Problem Solving.

53. Douglas B. Lenat.
AM: An Artificial Intelligence Approach to Discovery in Mathematics.

54. Douglas B. Lenat, Mayank Prakash, and Mary Shepherd.
Cyc: Using Common Sense Knowledge to Overcome Brittness and Knowledge Acquisition Bottlenecks.

A retrospective view of the Hearsay-II architecture.
In IJCAI-77 [48], page 790.

Wait-and-see strategies for parsing natural language.
57. Mitchell P. Marcus.
   *A Theory of Syntactic Recognition for Natural Language.*

58. David A. McAllester.
   Reasoning utility package user’s manual, version one.

   Some philosophical problems from the standpoint of artificial intelligence.
   In Meltzer and Michie [69], pages 463-502.

60. Drew V. McDermott.
   Assimilation of new information by a natural language-understanding system.

61. Drew V. McDermott.
   Contexts and data dependencies: a synthesis.

   *Flexibility and Efficiency in a Computer Program for Designing Circuits.*

63. Drew V. McDermott.
   The DUCK manual.

64. Drew V. McDermott.
   Vocabularies for problem-solver state descriptions.
   In IJCAI-77 [48], page 229.

65. Drew V. McDermott.
   Planning and acting.

66. Drew V. McDermott and Jon Doyle.
   Non-monotonic logic I.

67. Drew V. McDermott and Gerald Jay Sussman.
   *The CONNIVER Reference Manual.*

68. John McDermott and Charles L. Forgy.
   Production system conflict resolution strategies.
   In Waterman and Hayes-Roth [112], page 177.

69. Bernard Meltzer and Donald Michie, editors.
   *Machine Intelligence 4.*
70. B. Meltzer and D. Michie, editors.  
*Machine Intelligence* 7.  

71. D. Michie, editor.  
*Machine Intelligence* 3.  

72. Marvin Minsky.  
A framework for representing knowledge.  
In Winston [116], page 211.

73. Robert C. Moore.  
Reasoning from incomplete knowledge in a procedural deductive system.  

74. Thomas P. Moran.  
The symbolic nature of visual imagery.  
In IJCAI-73 [47], page 472.

75. Allen Newell.  
Production systems: models of control structures.  
In Chase [19], page 463.

76. Allen Newell.  
Artificial intelligence and the concept of mind.  
In Schank and Colby [95], page 1.

77. Allen Newell and Herbert A. Simon.  
*Human Problem Solving*.  

78. Allen Newell, editor.  
*Information Processing Language V Manual*.  

*Problem-Solving Methods in Artificial Intelligence*.  

80. M. S. Patterson and M. N. Wegman.  
Linear unification.  
*Proceedings of the Symposium on Theory of Complexity* 8.  

*The T Manual*.  

82. Jonathan Rees and William Clinger, editors.  
Revised³ Report on the Algorithmic Language *Scheme*.  
83. Charles J. Rieger.  
Spontaneous computation in cognitive models.  

84. Christopher K. Riesbeck.  
_Computational Understanding: Analysis of Sentences and Context._  
PhD dissertation, Stanford University, 1974.  
Stanford: AI Laboratory Memo 238.

85. R. B. Roberts and Ira P. Goldstein.  
_The FR1 Manual._  

86. J. A. Robinson.  
A machine-oriented logic based on the resolution principle.  

87. J. F. Rulifson.  
QA4 programming concepts.  

QA4: a procedural calculus for intuitive reasoning.  

89. R. Rustin, editor.  
_Natural Language Processing._  

90. Michael D. Rychener and Allen Newell.  
An instructable production system: basic design issues.  
In Waterman and Hayes-Roth [112], page 135.

91. Earl D. Sacerdoti.  
_A Structure for Plans and Behavior._  

92. Erik Sandewall.  
Ideas about management of LISP data bases.  

93. Roger C. Schank.  
_Consceptual Information Processing._  

_Scripts, Plans, Goals and Understanding._  

95. Roger C. Schank and Kenneth M. Colby, editors.  
_Computer Models of Thought and Language._  
96. Stuart C. Shapiro.
The SNePS semantic network processing system.
In Findler [36], pages 179-203.

*Computer-Based Medical Consultations: MYCIN.*

98. David C. Smith and Horace J. Enea.
Backtracking in MLISP2.
In IJCAI-73 [47], page 677.

Forward reasoning and dependency-directed backtracking in a system for
computer-aided circuit analysis.

100. Thomas A. Standish.
*Data Structure Techniques.*
Reading, Massachusetts: Addison-Wesley, 1980.

101. Guy L. Steele, Jr.
CLARIFICATIONS.TEXT.4, December 6, 1985.
Unpublished notes on recommended changes to COMMON LISP.
Distributed at the 1985 COMMON LISP meeting, Boston.

102. Guy L. Steele, Jr.
*COMMON LISP: The Language.*

103. Guy L. Steele, Jr.
Debunking the "expensive procedure call" myth, or Procedure call
implementations considered harmful, or LAMBDA: the ultimate goto.

104. Guy L. Steele, Jr.
*RABBIT: A Compiler for SCHEME (A Study in Compiler Optimization).*

105. Mark Stefik.
An examination of a frame-structured representation system.
In IJCAI-79 [49], pages 845-852.

106. Gerald Jay Sussman and Drew V. McDermott.
From PLANNER to CONNIVER — A genetic approach.

SCHEME: an interpreter for extended lambda calculus.
108. Gerald Jay Sussman and Guy L. Steele, Jr.
   The revised report on SCHEME.
   See also Rees and Clinger [82].


110. Warren Teitelman.

   The syntactic analysis of English by machine.
   In Michie [71], page 281.

112. Donald A. Waterman and Frederick Hayes-Roth, editors.
   Pattern-Directed Inference Systems.

113. Robert Wilensky.
   KODIAK — A Knowledge Representation Language.
   In COGSCI-84 [21], pages 344-352.

114. Terry Winograd.
   Understanding Natural Language.

115. Patrick H. Winston.
   Learning structural descriptions from examples.
   In Winston [116], page 157.

   The Psychology of Computer Vision.

117. Niklaus Wirth.
   Programming in MODULA-2.

118. William A. Woods.
   An experimental parsing system for transition network grammars.
   In Rustin [89], page 111.

119. R. M. Young.
   Mixtures of strategies in structurally adaptive production systems: examples from seriation and subtraction.