

# Paradigms of Artificial Intelligence Programming:

---

CASE STUDIES IN COMMON LISP

*Peter Norvig*



---

MORGAN KAUFMANN PUBLISHERS ◊ SAN FRANCISCO, CALIFORNIA

Sponsoring Editor *Michael B. Morgan*

Production Manager *Yonie Overton*

Cover Designer *Sandra Popovich*

Text Design/Composition *SuperScript Typography*

Copyeditor *Barbara Beidler Kendrick*

Proofreaders *Lynn Meinhardt, Sharilyn Hovind, Gary Morris*

Printer *Malloy Lithographing*

Morgan Kaufmann Publishers, Inc.

*Editorial and Sales Office:*

340 Pine Street, Sixth Floor

San Francisco, CA 94104-3205

USA

*Telephone* 415/392-2665

*Faxsimile* 415/982-2665

*Internet* mkp@mfp.com

*Web site* <http://mfp.com>

© 1992 Morgan Kaufmann Publishers, Inc.

All rights reserved

Printed in the United States of America

03 02 01 8 7 6

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means—electronic, photocopying, recording, or otherwise—with the prior written permission of the publisher.

Library of Congress Cataloging-in-Publication Data

Norvig, Peter.

Paradigms of artificial intelligence programming : case studies in common Lisp / Peter Norvig.

p. cm.

Includes bibliographical references and index.

ISBN 1-55860-191-0 :

1. Electronic digital computers—Programming. 2. COMMON LISP  
(Computer program language) 3. Artificial intelligence. I. Title.

QA76.6.N687 1991

006.3—dc20

91-39187

CIP

*To my family...*

# Preface

**paradigm** *n 1* an example or pattern; *esp* an outstandingly clear or typical example.

—*Longman's Dictionary of the English Language*, 1984

This book is concerned with three related topics: the field of artificial intelligence, or AI; the skill of computer programming; and the programming language Common Lisp. Careful readers of this book can expect to come away with an appreciation of the major questions and techniques of AI, an understanding of some important AI programs, and an ability to read, modify, and create programs using Common Lisp. The examples in this book are designed to be clear examples of good programming style—paradigms of programming. They are also paradigms of AI research—historically significant programs that use widely applicable techniques to solve important problems.

Just as a liberal arts education includes a course in “the great books” of a culture, so this book is, at one level, a course in “the great programs” that define the AI culture.<sup>1</sup>

At another level, this book is a highly technical compendium of the knowledge you will need to progress from being an intermediate Lisp programmer to being an expert. Parts I and II are designed to help the novice get up to speed, but the complete beginner may have a hard time even with this material. Fortunately, there are at least five good texts available for the beginner; see page xiii for my recommendations.

---

<sup>1</sup>This does not imply that the programs chosen are the best of all AI programs—just that they are representative.

All too often, the teaching of computer programming consists of explaining the syntax of the chosen language, showing the student a 10-line program, and then asking the student to write programs. In this book, we take the approach that the best way to learn to write is to read (and conversely, a good way to improve reading skills is to write). After the briefest of introductions to Lisp, we start right off with complex programs and ask the reader to understand and make small modifications to these programs.

The premise of this book is that you can only write something useful and interesting when you both understand what makes good writing and have something interesting to say. This holds for writing programs as well as for writing prose. As Kernighan and Plauger put it on the cover of *Software Tools in Pascal*:

*Good programming is not learned from generalities, but by seeing how significant programs can be made clean, easy to read, easy to maintain and modify, human-engineered, efficient, and reliable, by the application of common sense and good programming practices. Careful study and imitation of good programs leads to better writing.*

The proud craftsman is often tempted to display only the finished work, without any indication of the false starts and mistakes that are an unfortunate but unavoidable part of the creative process. Unfortunately, this reluctance to unveil the process is a barrier to learning; a student of mathematics who sees a beautiful 10-line proof in a textbook can marvel at its conciseness but does not learn how to construct such a proof. This book attempts to show the complete programming process, “warts and all.” Each chapter starts with a simple version of a program, one that works on some examples but fails on others. Each chapter shows how these failures can be analyzed to build increasingly sophisticated versions of the basic program. Thus, the reader can not only appreciate the final result but also see how to learn from mistakes and refine an initially incomplete design. Furthermore, the reader who finds a particular chapter is becoming too difficult can skip to the next chapter, having gained some appreciation of the problem area, and without being overwhelmed by the details.

This book presents a body of knowledge loosely known as “AI programming techniques,” but it must be recognized that there are no clear-cut boundaries on this body of knowledge. To be sure, no one can be a good AI programmer without first being a good programmer. Thus, this book presents topics (especially in parts III and V) that are not AI per se, but are essential background for any AI practitioner.

### *Why Lisp? Why Common Lisp?*

Lisp is one of the oldest programming languages still in widespread use today. There have been many versions of Lisp, each sharing basic features but differing in detail. In this book we use the version called Common Lisp, which is the most widely accepted standard. Lisp has been chosen for three reasons.

First, Lisp is the most popular language for AI programming, particularly in the United States. If you're going to learn a language, it might as well be one with a growing literature, rather than a dead tongue.

Second, Lisp makes it easy to capture relevant generalizations in defining new objects. In particular, Lisp makes it easy to define new languages especially targeted to the problem at hand. This is especially handy in AI applications, which often manipulate complex information that is most easily represented in some novel form. Lisp is one of the few languages that allows full flexibility in defining and manipulating programs as well as data. All programming languages, by definition, provide a means of defining programs, but many other languages limit the ways in which a program can be used, or limit the range of programs that can be defined, or require the programmer to explicitly state irrelevant details.

Third, Lisp makes it very easy to develop a working program fast. Lisp programs are concise and are uncluttered by low-level detail. Common Lisp offers an unusually large number of useful predefined objects, including over 700 functions. The programming environment (such as debugging tools, incremental compilers, integrated editors, and interfaces to window systems) that surround Lisp systems are usually very good. And the dynamic, interactive nature of Lisp makes it easy to experiment and change a program while it is being developed.

It must be mentioned that in Europe and Japan, Prolog has been as popular as Lisp for AI work. Prolog shares most of Lisp's advantages in terms of flexibility and conciseness. Recently, Lisp has gained popularity worldwide, and Prolog is becoming more well known in the United States. As a result, the average AI worker today is likely to be bilingual. This book presents the key ideas behind Prolog in chapters 11 and 12, and uses these ideas in subsequent chapters, particularly 20 and 21.

The dialect of Lisp known as Scheme is also gaining in popularity, but primarily for teaching and experimenting with programming language design and techniques, and not so much for writing large AI programs. Scheme is presented in chapters 22 and 23. Other dialects of Lisp such as Franz Lisp, MacLisp, InterLisp, ZetaLisp, and Standard Lisp are now considered obsolete. The only new dialect of Lisp to be proposed recently is EuLisp, the European Lisp. A few dialects of Lisp live on as embedded extension languages. For example, the Gnu Emacs text editor uses elisp, and the AutoCad computer-aided design package uses AutoLisp, a derivative of Xlisp. In the future, it is likely that Scheme will become a popular extension language, since it is small but powerful and has an officially sanctioned standard definition.

There is a myth that Lisp (and Prolog) are "special-purpose" languages, while languages like Pascal and C are "general purpose." Actually, just the reverse is true. Pascal and C are special-purpose languages for manipulating the registers and memory of a von Neumann-style computer. The majority of their syntax is devoted to arithmetic and Boolean expressions, and while they provide some facilities for forming data structures, they have poor mechanisms for procedural abstraction or control abstraction. In addition, they are designed for the state-oriented style

of programming: computing a result by changing the value of variables through assignment statements.

Lisp, on the other hand, has no special syntax for arithmetic. Addition and multiplication are no more or less basic than list operations like appending, or string operations like converting to upper case. But Lisp provides all you will need for programming in general: defining data structures, functions, and the means for combining them.

The assignment-dominated, state-oriented style of programming is possible in Lisp, but in addition object-oriented, rule-based, and functional styles are all supported within Lisp. This flexibility derives from two key features of Lisp: First, Lisp has a powerful *macro* facility, which can be used to extend the basic language. When new styles of programming were invented, other languages died out; Lisp simply incorporated the new styles by defining some new macros. The macro facility is possible because Lisp programs are composed of a simple data structure: the list. In the early days, when Lisp was interpreted, most manipulation of programs was done through this data structure. Nowadays, Lisp is more often compiled than interpreted, and programmers rely more on Lisp's second great flexible feature: the *function*. Of course, other languages have functions, but Lisp is rare in allowing the creation of new functions while a program is running.

Lisp's flexibility allows it to adapt as programming styles change, but more importantly, Lisp can adapt to your particular programming problem. In other languages you fit your problem to the language; with Lisp you extend the language to fit your problem.

Because of its flexibility, Lisp has been successful as a high-level language for rapid prototyping in areas such as AI, graphics, and user interfaces. Lisp has also been the dominant language for exploratory programming, where the problems are so complex that no clear solution is available at the start of the project. Much of AI falls under this heading.

The size of Common Lisp can be either an advantage or a disadvantage, depending on your outlook. In David Touretzky's (1989) fine book for beginning programmers, the emphasis is on simplicity. He chooses to write some programs slightly less concisely, rather than introduce an esoteric new feature (he cites `pushnew` as an example). That approach is entirely appropriate for beginners, but this book goes well past the level of beginner. This means exposing the reader to new features of the language whenever they are appropriate. Most of the time, new features are described as they are introduced, but sometimes explaining the details of a low-level function would detract from the explanation of the workings of a program. In accepting the privilege of being treated as an "adult," the reader also accepts a responsibility—to look up unfamiliar terms in an appropriate reference source.

## *Outline of the Book*

This book is organized into five parts.

**Part I** introduces the Common Lisp programming language.

Chapter 1 gives a quick introduction by way of small examples that demonstrate the novel features of Lisp. It can be safely skipped or skimmed by the experienced programmer.

Chapter 2 is a more extended example showing how the Lisp primitives can be put together to form a program. It should be studied carefully by the novice, and even the experienced programmer will want to look through it to get a feel for my programming style.

Chapter 3 provides an overview of the Lisp primitives. It can be skimmed on first reading and used as a reference whenever an unfamiliar function is mentioned in the text.

Part I has been kept intentionally brief, so that there is more room for presenting actual AI programs. Unfortunately, that means that another text or reference book (or online help) may be needed to clarify some of the more esoteric features of the language. My recommendations for texts are on page xiii.

The reader may also want to refer to chapter 25, which offers some debugging and troubleshooting hints.

**Part II** covers four early AI programs that all use rule-based pattern-matching techniques. By starting with relatively simple versions of the programs and then improving them and moving on to more complex programs, the reader is able to gradually acquire increasingly advanced programming skills.

Chapter 4 presents a reconstruction of GPS, the General Problem Solver. The implementation follows the STRIPS approach.

Chapter 5 describes ELIZA, a program that mimics human dialogue. This is followed by a chapter that generalizes some of the techniques used in GPS and ELIZA and makes them available as tools for use in subsequent programs.

Chapter 7 covers STUDENT, a program that solves high-school-level algebra word problems.

Chapter 8 develops a small subset of the MACSYMA program for doing symbolic algebra, including differential and integral calculus. It may be skipped by those who shy away from heavy mathematics.

**Part III** detours from AI for a moment to present some general tools for more efficient programming. The reader who masters the material in this part can be considered an advanced Lisp programmer.

Chapter 9 is a detailed study of efficiency techniques, concentrating on caching, indexing, compilation, and delaying computation. Chapter 10 covers lower-level efficiency issues such as using declarations, avoiding garbage generation, and choosing the right data structure.

Chapter 11 presents the Prolog language. The aim is two-fold: to show how to write an interpreter for another language, and to introduce the important features of Prolog, so that they can be used where appropriate. Chapter 12 shows how a compiler for Prolog can be 20 to 200 times faster than the interpreter.

Chapter 13 introduces object-oriented programming in general, then explores the Common Lisp Object System (CLOS).

Chapter 14 discusses the advantages and limitations of both logic-oriented and object-oriented programming, and develops a knowledge representation formalism using all the techniques of part III.

**Part IV** covers some advanced AI programs.

Chapter 15 uses the techniques of part III to come up with a much more efficient implementation of MACSYMA. It uses the idea of a canonical form, and replaces the very general rewrite rule approach with a series of more specific functions.

Chapter 16 covers the EMYCIN expert system shell, a backward chaining rule-based system based on certainty factors. The MYCIN medical expert system is also covered briefly.

Chapter 17 covers the Waltz line-labeling algorithm for polyhedra (using Huffman-Clowes labels). Different approaches to constraint propagation and backtracking are discussed.

Chapter 18 presents a program that plays an excellent game of Othello. The technique used, alpha-beta searching, is appropriate to a wide variety of two-person games.

Chapter 19 is an introduction to natural language processing. It covers context-free grammar, top-down and bottom-up parsing, chart parsing, and some semantic interpretation and preferences.

Chapter 20 extends the linguistic coverage of the previous chapter and introduces logic grammars, using the Prolog compiler developed in chapter 11.

Chapter 21 is a fairly comprehensive grammar of English using the logic grammar formalism. The problems of going from a simple idea to a realistic, comprehensive program are discussed.

**Part V** includes material that is peripheral to AI but important for any serious Lisp programmer.

Chapter 22 presents the Scheme dialect of Lisp. A simple Scheme interpreter is developed, then a properly tail-recursive interpreter, then an interpreter that explicitly manipulates continuations and supports `call/cc`. Chapter 23 presents a Scheme compiler.

Chapter 24 presents the features that are unique to American National Standards Institute (ANSI) Common Lisp. This includes the `loop` macro, as well as error handling, pretty printing, series and sequences, and the package facility.

Chapter 25 is a guide to troubleshooting and debugging Lisp programs.

The bibliography lists over 200 sources, and there is a comprehensive index. In addition, the appendix provides a directory of publicly available Lisp programs.

### *How to Use This Book*

The intended audience for this book is broad: anyone who wants to become an advanced Lisp programmer, and anyone who wants to be an advanced AI practitioner. There are several recommended paths through the book:

- *In an Introductory AI Course:* Concentrate on parts I and II, and at least one example from part IV.
- *In an Advanced AI Programming Course:* Concentrate on parts I, II and IV, skipping chapters that are of less interest and adding as much of part III as time permits.
- *In an Advanced Programming Languages Course:* Concentrate on parts I and V, with selections from part III. Cover chapters 11 and 13 if similar material is not presented with another text.
- *For the Professional Lisp Programmer:* Read as much of the book as possible, and refer back to it often. Part III and chapter 25 are particularly important.

### *Supplementary Texts and Reference Books*

The definitive reference source is Steele's *Common Lisp the Language*. From 1984 to 1990, this unambiguously defined the language Common Lisp. However, in 1990 the picture became more complicated by the publication of *Common Lisp the Language*, 2d edition. This book, also by Steele, contains the recommendations of ANSI subcommittee X3J13, whose charter is to define a standard for Lisp. These recommendations include many minor changes and clarifications, as well as brand new material on object-oriented programming, error condition handling, and the loop macro. The new material doubles the size of the book from 465 to 1029 pages.

Until the ANSI recommendations are formally accepted, Common Lisp users are in the unfortunate situation of having two distinct and incompatible standards: "original" Common Lisp and ANSI Common Lisp. Most of the code in this book is compliant with both standards. The most significant use of an ANSI function is the loop macro. The ANSI map-into, complement, and reduce functions are also used, although rarely. Definitions for all these functions are included, so even those using an "original" Common Lisp system can still run all the code in the book.

While *Common Lisp the Language* is the definitive standard, it is sometimes terse and can be difficult for a beginner. *Common Lisp: the Reference*, published by Franz Inc., offers complete coverage of the language with many helpful examples. *Common LISPcraft*, by Robert Wilensky, and *Artificial Intelligence Programming*, by Charniak

et al., also include brief summaries of the Common Lisp functions. They are not as comprehensive, but that can be a blessing, because it can lead the reader more directly to the functions that are important (at least in the eyes of the author).

It is a good idea to read this book with a computer at hand, to try out the examples and experiment with examples of your own. A computer is also handy because Lisp is self-documenting, through the functions `apropos`, `describe`, and `documentation`. Many implementations also provide more extensive documentation through some kind of ‘help’ command or menu.

The five introductory Lisp textbooks I recommend are listed below. The first is more elementary than the others.

- *Common Lisp: A Gentle Introduction to Symbolic Computation* by David Touretzky. Most appropriate for beginners, including those who are not computer scientists.
- *A Programmer’s Guide to Common Lisp* by Deborah G. Tatar. Appropriate for those with experience in another programming language, but none in Lisp.
- *Common LISPcraft* by Robert Wilensky. More comprehensive and faster paced, but still useful as an introduction as well as a reference.
- *Common Lisp* by Wade L. Hennessy. Somewhat hit-and-miss in terms of the topics it covers, but with an enlightened discussion of implementation and efficiency issues that do not appear in the other texts.
- *LISP (3d edition)* by Patrick H. Winston and Bertold Horn. Covers the most ground in terms of programming advice, but not as comprehensive as a reference. May be difficult for beginners. Includes some AI examples.

While it may be distracting for the beginner to be continually looking at some reference source, the alternative—to have this book explain every new function in complete detail as it is introduced—would be even more distracting. It would interrupt the description of the AI programs, which is what this book is all about.

There are a few texts that show how to write AI programs and tools, but none that go into the depth of this book. Nevertheless, the expert AI programmer will want to be familiar with all the following texts, listed in rough order of increasing sophistication:

- *LISP (3d edition)*. (See above.)
- *Programming Paradigms in Lisp* by Rajeev Sangal. Presents the different styles of programming that Lisp accommodates, illustrating them with some useful AI tools.

- *Programming for Artificial Intelligence* by Wolfgang Kreutzer and Bruce McKenzie. Covers some of the basics of rule-based and pattern-matching systems well, but covers Lisp, Prolog, and Smalltalk, and thus has no time left for details in any of the languages.
- *Artificial Intelligence Programming* (2d edition) by Eugene Charniak, Christopher Riesbeck, Drew McDermott, and James Meehan. Contains 150 pages of Lisp overview, followed by an advanced discussion of AI tools, but no actual AI programs.
- *AI in Practice: Examples in Pop-11* by Allan Ramsey and Rosalind Barrett. Advanced, high-quality implementations of five AI programs, unfortunately using a language that has not gained popularity.

The current text combines the virtues of the last two entries: it presents both actual AI programs and the tools necessary to build them. Furthermore, the presentation is in an incremental fashion, with simple versions presented first for clarity, followed by more sophisticated versions for completeness.

### *A Note on Exercises*

Sample exercises are provided throughout. Readers can test their level of understanding by faithfully doing the exercises. The exercises are graded on the scale [s], [m], [h], [d], which can be interpreted either as a level of difficulty or as an expected time it will take to do the exercise:

Code	Difficulty	Time to Do
[s]	Simple	Seconds
[m]	Medium	Minutes
[h]	Hard	Hours
[d]	Difficult	Days

The time to do the exercise is measured from the point that the concepts have been well understood. If the reader is unclear on the underlying concepts, it might take hours of review to understand a [m] problem. Answers to the exercises can be found in a separate section at the end of each chapter.

### *Acknowledgments*

A great many people contributed to this book. First of all I would like to thank my students at USC and Berkeley, as well as James Martin's students at Colorado and Michael Pazzani's students at Irvine, who course-tested earlier versions of this book. Useful suggestions, corrections, and additions were made by:

Nina Amenta (Berkeley), Ray S. Babcock and John Paxton (Montana State), Bryan A. Bentz (BBN), Mary P. Boelk (Johnson Controls), Michael Braverman (Berkeley), R. Chandrasekar and M. Sasikumar (National Centre for Software Technology, Bombay), Mike Clancy (Berkeley), Michael Covington (Georgia), Bruce D'Ambrosio (Oregon State), Piew Datta (Irvine), Shawn Dettrey (USC), J. A. Durieux (AI Engineering BV, Amsterdam), Joseph Faletti (ETS), Paul Fuqua (Texas Instruments), Robert Goldman (Tulane), Marty Hall (Johns Hopkins), Marti Hearst (Berkeley), Jim Helder (Maryland), Phil Laird (NASA), Raymond Lang (Tulane), David D. Loefler (MCC), George Luger (New Mexico), Rob MacLachlan (CMU), Barry Margolin (Thinking Machines), James Mayfield (UMBC), Sanjay Manchandi (Arizona), Robert McCartney (Connecticut), James Meehan (DEC), Andrew L. Ressler, Robert S. Rist (University of Technology, Sydney), Paul Snively (Apple), Peter Van Roy (Berkeley), David Gumby Wallace (Cygnus), and Jeff Wu (Colorado).

Sam Dooley and Eric Wefald both wrote Othello-playing programs without which I would not have written chapter 18. Eric also showed me Aristotle's quotes on means-ends analysis. Tragically, Eric died in August 1989. He is sorely missed by his friends and colleagues. Richard Fateman made suggestions for chapter 8, convinced me to write chapter 15, and, with help from Peter Klier, wrote a substantial program from which I adapted some code for that chapter. Charley Cox (Franz Inc.), Jamie Zawinski (Lucid Inc.), and Paul Fuqua (Texas Instruments) explained the inner workings of their respective companies' compilers. Mike Harrison, Paul Hilfinger, Marc Luria, Ethan Munson, and Stephan Slade helped with L<sup>A</sup>T<sub>E</sub>X. Narciso Jarimillo tested all the code and separated it into the files that are available to the reader (see page 897).

During the writing of this book I was supported by a grant from the Defense Advanced Research Projects Agency (DoD), Arpa Order No. 4871, monitored by Space and Naval Warfare Systems Command under Contract N00039-84-C-0089. Special thanks to DARPA and to Robert Wilensky and the rest of my colleagues and students at Berkeley for providing a stimulating environment for research, programming, and writing.

Finally, thanks to Mike Morgan and Yonie Overton for overseeing the production of the book and encouraging me to finish on time.

## CHAPTER 1

---

# Introduction to Lisp

*You think you know when you learn, are more sure  
when you can write, even more when you can teach,  
but certain when you can program.*

—Alan Perlis  
Yale University computer scientist

This chapter is for people with little or no experience in Lisp. Readers who feel confident in their Lisp programming ability can quickly skim the chapter or skip it entirely. This chapter necessarily moves quickly, so those with little programming experience, or any reader who finds this chapter tough going, should seek out a supplementary introductory text. My recommendations are in the preface.

Computers allow one to carry out computations. A word processing program deals with words while a calculator deals with numbers, but the principles are the same. In both cases, you provide the input (words or numbers) and specify the operations (such as deleting a word or adding two numbers) to yield a result (a completed document or calculation).

We will refer to anything that can be represented in the memory of a computer as a *computational object*, or just an *object*. So, words, paragraphs, and numbers can be objects. And because the operations (deleting and adding) must be represented somewhere in the computer's memory, they are objects, too.

Normally, the distinction between a computer “user” and a computer “programmer” is that the user provides new input, or data (words or numbers), while the programmer defines new *operations*, or programs, as well as new *types* of data. Every new object, be it datum or operation, must be defined in terms of previously defined objects. The bad news is that it can be quite tedious to get these definitions right. The good news is that each new object can in turn be used in the definition of future objects. Thus, even complex programs can be built out of smaller, simpler objects. This book covers a number of typical AI problems, showing how each problem can be broken down into manageable pieces, and also how each piece can be described in the programming language Common Lisp. Ideally, readers will learn enough through studying these examples to attack new AI problems with style, grace, and success.

Let’s consider a simple example of a computation: finding the sum of two numbers, let’s say 2 and 2. If we had a calculator handy, we would type “ $2 + 2 =$ ” and see the answer displayed. On a calculator using reverse Polish notation, we would have to type “ $2\ 2\ +$ ” to see the same answer. In Lisp, as with the calculator, the user carries out an interactive dialog with the computer by typing in an expression and seeing the computer print the value of that expression. This interactive mode is different from many other programming languages that only offer a batch mode, wherein an entire program is compiled and run before any output can be seen.

We start up a pocket calculator by flipping the on/off switch. The Lisp program must also be started, but the details vary from one computer to another, so I can’t explain how your Lisp will work. Assuming we have managed to start up Lisp, we are likely to see a *prompt* of some kind. On my computer, Lisp types “>” to indicate it is ready to accept the next computation. So we are faced with a screen that looks like this:

>

We may now type in our computation and see the result displayed. It turns out that the Lisp convention for arithmetic expressions is slightly different: a computation consists of a parenthesized list with the operation name first, followed by any number of operands, or arguments. This is called *prefix notation*.

```
> (+ 2 2)
4
>
```

We see that Lisp has printed the answer, 4, and then another prompt, >, to indicate it is ready for the next computation. Throughout this book, all Lisp expressions will be displayed in typewriter font. Text on the same line as the “>” prompt is input typed by the user, and text following it is output printed by the computer. Usually, input that is typed by the programmer will be in lowercase letters, while output that

is printed back by the computer will be in UPPERCASE letters. Of course, with symbols like + and 4 there is no difference.

To save space on the page, the output will sometimes be shown on the same line as the input, separated by an arrow ( $\Rightarrow$ ), which can be read as “evaluates to,” and can also be thought of as standing for the return or enter key that the user presses to complete the input:

```
> (+ 2 2) ⇒ 4
```

One advantage of parenthesized prefix notation is that the parentheses clearly mark the beginning and end of an expression. If we want, we can give + more than two arguments, and it will still add them all:

```
> (+ 1 2 3 4 5 6 7 8 9 10) ⇒ 55
```

This time we try  $(9000 + 900 + 90 + 9) - (5000 + 500 + 50 + 5)$ :

```
> (- (+ 9000 900 90 9) (+ 5000 500 50 5)) ⇒ 4444
```

This example shows that expressions can be nested. The arguments to the - function are parenthesized lists, while the arguments to each + are atoms. The Lisp notation may look unusual compared to standard mathematical notation, but there are advantages to this notation; since Lisp expressions can consist of a function followed by any number of arguments, we don’t have to keep repeating the “+.” More important than the notation is the rule for evaluation. In Lisp, lists are evaluated by first evaluating all the arguments, then applying the function to the arguments, thereby computing the result. This rule is much simpler than the rule for evaluating normal mathematical expressions, where there are many conventions to remember, such as doing multiplications and divisions before sums and differences. We will see below that the actual Lisp evaluation rule is a little more complicated, but not much.

Sometimes programmers who are familiar with other languages have preconceptions that make it difficult for them to learn Lisp. For them, three points are worth stressing here. First, many other languages make a distinction between statements and expressions. An expression, like  $2 + 2$ , has a value, but a statement, like  $x = 2 + 2$ , does not. Statements have effects, but they do not return values. In Lisp, there is no such distinction: every expression returns a value. It is true that some expressions have effects, but even those expressions also return values.

Second, the lexical rules for Lisp are much simpler than the rules for other languages. In particular, there are fewer punctuation characters: only parentheses, quote marks (single, double, and backward), spaces, and the comma serve to separate symbols from each other. Thus, while the statement  $y=a*x+3$  is analyzed as seven separate tokens in other languages, in Lisp it would be treated as a single symbol. To

get a list of tokens, we would have to insert spaces: ( $y = a * x + 3$ ).<sup>1</sup>

Third, while many languages use semicolons to delimit statements, Lisp has no need of semicolons, since expressions are delimited by parentheses. Lisp chooses to use semicolons for another purpose—to mark the beginning of a comment, which lasts until the end of the line:

```
> (+ 2 2) ; this is a comment
4
```

## 1.1 Symbolic Computation

All we've done so far is manipulate numbers in the same way a simple pocket calculator would. Lisp is more useful than a calculator for two main reasons. First, it allows us to manipulate objects other than numbers, and second, it allows us to define new objects that might be useful in subsequent computations. We will examine these two important properties in turn.

Besides numbers, Lisp can represent characters (letters), strings of characters, and arbitrary symbols, where we are free to interpret these symbols as referring to things outside the world of mathematics. Lisp can also build nonatomic objects by combining several objects into a list. This capability is fundamental and well supported in the language; in fact, the name Lisp is short for LISt Processing.

Here's an example of a computation on lists:

```
> (append '(Pat Kim) '(Robin Sandy)) ⇒ (PAT KIM ROBIN SANDY)
```

This expression appends together two lists of names. The rule for evaluating this expression is the same as the rule for numeric calculations: apply the function (in this case append) to the value of the arguments.

The unusual part is the quote mark ('), which serves to block the evaluation of the following expression, returning it literally. If we just had the expression (Pat Kim), it would be evaluated by considering Pat as a function and applying it to the value of the expression Kim. This is not what we had in mind. The quote mark instructs Lisp to treat the list as a piece of data rather than as a function call:

```
> '(Pat Kim) ⇒ (PAT KIM)
```

In other computer languages (and in English), quotes usually come in pairs: one to mark the beginning, and one to mark the end. In Lisp, a single quote is used to mark

---

<sup>1</sup>This list of symbols is not a legal Lisp assignment statement, but it is a Lisp data object.

the beginning of an expression. Since we always know how long a single expression is—either to the end of an atom or to the matching parenthesis of a list—we don't need an explicit punctuation mark to tell us where the expression ends. Quotes can be used on lists, as in '(Pat Kim), on symbols as in 'Robin, and in fact on anything else. Here are some examples:

```
> 'John => JOHN
> '(John Q Public) => (JOHN Q PUBLIC)
> '2 => 2
> 2 => 2
> '(+ 2 2) => (+ 2 2)
> (+ 2 2) => 4
> John => Error: JOHN is not a bound variable
> (John Q Public) => Error: JOHN is not a function
```

Note that '2 evaluates to 2 because it is a quoted expression, and 2 evaluates to 2 because numbers evaluate to themselves. Same result, different reason. In contrast, 'John evaluates to John because it is a quoted expression, but evaluating John leads to an error, because evaluating a symbol means getting the value of the symbol, and no value has been assigned to John.

Symbolic computations can be nested and even mixed with numeric computations. The following expression builds a list of names in a slightly different way than we saw before, using the built-in function `list`. We then see how to find the number of elements in the list, using the built-in function `length`:

```
> (append '(Pat Kim) (list '(John Q Public) 'Sandy))
(PAT KIM (JOHN Q PUBLIC) SANDY)
> (length (append '(Pat Kim) (list '(John Q Public) 'Sandy)))
4
```

There are four important points to make about symbols:

- First, it is important to remember that Lisp does not attach any external significance to the objects it manipulates. For example, we naturally think of (Robin Sandy) as a list of two first names, and (John Q Public) as a list of one person's first name, middle initial, and last name. Lisp has no such preconceptions. To Lisp, both Robin and xyzzy are perfectly good symbols.
- Second, to do the computations above, we had to know that `append`, `length`, and `+` are defined functions in Common Lisp. Learning a language involves

remembering vocabulary items (or knowing where to look them up) as well as learning the basic rules for forming expressions and determining what they mean. Common Lisp provides over 700 built-in functions. At some point the reader should flip through a reference text to see what's there, but most of the important functions are presented in part I of this book.

- Third, note that symbols in Common Lisp are not case sensitive. By that I mean that the inputs John, john, and j0hN all refer to the same symbol, which is normally printed as JOHN.<sup>2</sup>
- Fourth, note that a wide variety of characters are allowed in symbols: numbers, letters, and other punctuation marks like '+' or '!'. The exact rules for what constitutes a symbol are a little complicated, but the normal convention is to use symbols consisting mostly of letters, with words separated by a dash (-), and perhaps with a number at the end. Some programmers are more liberal in naming variables, and include characters like '?!\$/<=>'. For example, a function to convert dollars to yen might be named with the symbol \$-to-yen or \$->yen in Lisp, while one would use something like DollarsToYen, dollars\_to\_yen or dol2yen in Pascal or C. There are a few exceptions to these naming conventions, which will be dealt with as they come up.

## 1.2 Variables

We have seen some of the basics of symbolic computation. Now we move on to perhaps the most important characteristic of a programming language: the ability to define new objects in terms of others, and to name these objects for future use. Here symbols again play an important role—they are used to name variables. A variable can take on a value, which can be any Lisp object. One way to give a value to a variable is with setf:

```
> (setf p '(John Q Public)) => (JOHN Q PUBLIC)
> p => (JOHN Q PUBLIC)
> (setf x 10) => 10
> (+ x x) => 20
> (+ x (length p)) => 13
```

After assigning the value (John Q Public) to the variable named p, we can refer to the value with the name p. Similarly, after assigning a value to the variable named x, we can refer to both x and p.

---

<sup>2</sup>The variable \*print-case\* controls how symbols will be printed. By default, the value of this variable is :upcase, but it can be changed to :downcase or :capitalize.

Symbols are also used to name functions in Common Lisp. Every symbol can be used as the name of a variable or a function, or both, although it is rare (and potentially confusing) to have symbols name both. For example, `append` and `length` are symbols that name functions but have no values as variables, and `pi` does not name a function but is a variable whose value is `3.1415926535897936` (or thereabout).

## 1.3 Special Forms

The careful reader will note that `setf` violates the evaluation rule. We said earlier that functions like `+`, `-` and `append` work by first evaluating all their arguments and then applying the function to the result. But `setf` doesn't follow that rule, because `setf` is not a function at all. Rather, it is part of the basic syntax of Lisp. Besides the syntax of atoms and function calls, Lisp has a small number of syntactic expressions. They are known as *special forms*. They serve the same purpose as statements in other programming languages, and indeed have some of the same syntactic markers, such as `if` and `loop`. There are two main differences between Lisp's syntax and other languages. First, Lisp's syntactic forms are always lists in which the first element is one of a small number of privileged symbols. `setf` is one of these symbols, so `(setf x 10)` is a special form. Second, special forms are expressions that return a value. This is in contrast to statements in most languages, which have an effect but do not return a value.

In evaluating an expression like `(setf x (+ 1 2))`, we set the variable named by the symbol `x` to the value of `(+ 1 2)`, which is 3. If `setf` were a normal function, we would evaluate both the symbol `x` and the expression `(+ 1 2)` and do something with these two values, which is not what we want at all. `setf` is called a special form because it does something special: if it did not exist, it would be impossible to write a function that assigns a value to a variable. The philosophy of Lisp is to provide a small number of special forms to do the things that could not otherwise be done, and then to expect the user to write everything else as functions.

The term *special form* is used confusingly to refer both to symbols like `setf` and expressions that start with them, like `(setf x 3)`. In the book *Common LISPcraft*, Wilensky resolves the ambiguity by calling `setf` a *special function*, and reserving the term *special form* for `(setf x 3)`. This terminology implies that `setf` is just another function, but a special one in that its first argument is not evaluated. Such a view made sense in the days when Lisp was primarily an interpreted language. The modern view is that `setf` should not be considered some kind of abnormal function but rather a marker of special syntax that will be handled specially by the compiler. Thus, the special form `(setf x (+ 2 1))` should be considered the equivalent of `x = 2 + 1` in C. When there is risk of confusion, we will call `setf` a *special form operator* and `(setf x 3)` a *special form expression*.

It turns out that the quote mark is just an abbreviation for another special form. The expression '*x*' is equivalent to '(*quote x*)', a special form expression that evaluates to *x*. The special form operators used in this chapter are:

defun	define function
defparameter	define special variable
setf	set variable or field to new value
let	bind local variable(s)
case	choose one of several alternatives
if	do one thing or another, depending on a test
function (#')	refer to a function
quote (')	introduce constant data

## 1.4 Lists

So far we have seen two functions that operate on lists: `append` and `length`. Since lists are important, let's look at some more list processing functions:

```
> p => (JOHN Q PUBLIC)
> (first p) => JOHN
> (rest p) => (Q PUBLIC)
> (second p) => Q
> (third p) => PUBLIC
> (fourth p) => NIL
> (length p) => 3
```

The functions `first`, `second`, `third`, and `fourth` are aptly named: `first` returns the first element of a list, `second` gives you the second element, and so on. The function `rest` is not as obvious; its name stands for "the rest of the list after the first element." The symbol `nil` and the form `()` are completely synonymous; they are both representations of the empty list. `nil` is also used to denote the "false" value in Lisp. Thus, `(fourth p)` is `nil` because there is no fourth element of `p`. Note that lists need not be composed only of atoms, but can contain sublists as elements:

```
> (setf x '((1st element) 2 (element 3) ((4)) 5))
((1ST ELEMENT) 2 (ELEMENT 3) ((4)) 5)
> (length x) => 5
> (first x) => (1ST ELEMENT)
```

---

```

> (second x) => 2
> (third x) => (ELEMENT 3)
> (fourth x) => ((4))
> (first (fourth x)) => (4)
> (first (first (fourth x))) => 4
> (fifth x) => 5
> (first x) => (1ST ELEMENT)
> (second (first x)) => ELEMENT

```

So far we have seen how to access parts of lists. It is also possible to build up new lists, as these examples show:

```

> p => (JOHN Q PUBLIC)
> (cons 'Mr p) => (MR JOHN Q PUBLIC)
> (cons (first p) (rest p)) => (JOHN Q PUBLIC)
> (setf town (list 'Anytown 'USA)) => (ANYTOWN USA)
> (list p 'of town 'may 'have 'already 'won!) =>
((JOHN Q PUBLIC) OF (ANYTOWN USA) MAY HAVE ALREADY WON!)
> (append p '(of) town '(may have already won!)) =>
(JOHN Q PUBLIC OF ANYTOWN USA MAY HAVE ALREADY WON!)
> p => (JOHN Q PUBLIC)

```

The function `cons` stands for “construct.” It takes as arguments an element and a list,<sup>3</sup> and constructs a new list whose first is the element and whose rest is the original list. `list` takes any number of elements as arguments and returns a new list containing those elements in order. We’ve already seen `append`, which is similar to `list`; it takes as arguments any number of lists and appends them all together, forming one big list. Thus, the arguments to `append` must be lists, while the arguments to `list` may be lists or atoms. It is important to note that these functions create new lists; they don’t modify old ones. When we say `(append p q)`, the effect is to create a brand new list that starts with the same elements that were in `p`. `p` itself remains unchanged.

Now let’s move away from abstract functions on lists, and consider a simple problem: given a person’s name in the form of a list, how might we extract the family name? For `(JOHN Q PUBLIC)` we could just use the function `third`, but that wouldn’t

---

<sup>3</sup>Later we will see what happens when the second argument is not a list.

work for someone with no middle name. There is a function called `last` in Common Lisp; perhaps that would work. We can experiment:

```
> (last p) => (PUBLIC)  
> (first (last p)) => PUBLIC
```

It turns out that `last` perversely returns a list of the last element, rather than the last element itself.<sup>4</sup> Thus we need to combine `first` and `last` to pick out the actual last element. We would like to be able to save the work we've done, and give it a proper description, like `last-name`. We could use `setf` to save the last name of `p`, but that wouldn't help determine any other last name. Instead we want to define a new function that computes the last name of *any* name that is represented as a list. The next section does just that.

## 1.5 Defining New Functions

The special form `defun` stands for “define function.” It is used here to define a new function called `last-name`:

```
(defun last-name (name)  
  "Select the last name from a name represented as a list."  
  (first (last name)))
```

We give our new function the name `last-name`. It has a *parameter list* consisting of a single parameter: `(name)`. This means that the function takes one argument, which we will refer to as `name`. It also has a *documentation string* that states what the function does. This is not used in any computation, but documentation strings are crucial tools for debugging and understanding large systems. The body of the definition is `(first (last name))`, which is what we used before to pick out the last name of `p`. The difference is that here we want to pick out the last name of *any* name, not just of the particular name `p`.

In general, a function definition takes the following form (where the documentation string is optional, and all other parts are required):

---

<sup>4</sup>In ANSI Common Lisp, `last` is defined to return a list of the last  $n$  elements, where  $n$  defaults to 1. Thus  $(\text{last } p) \equiv (\text{last } p \ 1) = (\text{PUBLIC})$ , and  $(\text{last } p \ 2) = (Q \ \text{PUBLIC})$ . This may make the definition of `last` seem less perverse.

---

```
(defun function-name (parameter...)
  "documentation string"
  function-body...)
```

The function name must be a symbol, the parameters are usually symbols (with some complications to be explained later), and the function body consists of one or more expressions that are evaluated when the function is called. The last expression is returned as the value of the function call.

Once we have defined `last-name`, we can use it just like any other Lisp function:

```
> (last-name p) => PUBLIC
> (last-name '(Rear Admiral Grace Murray Hopper)) => HOPPER
> (last-name '(Rex Morgan MD)) => MD
> (last-name '(Spot)) => SPOT
> (last-name '(Aristotle)) => ARISTOTLE
```

The last three examples point out an inherent limitation of the programming enterprise. When we say `(defun last-name...)` we are not really defining what it means for a person to have a last name; we are just defining an operation on a representation of names in terms of lists. Our intuitions—that MD is a title, Spot is the first name of a dog, and Aristotle lived before the concept of last name was invented—are not represented in this operation. However, we could always change the definition of `last-name` to incorporate these problematic cases.

We can also define the function `first-name`. Even though the definition is trivial (it is the same as the function `first`), it is still good practice to define `first-name` explicitly. Then we can use the function `first-name` when we are dealing with names, and `first` when we are dealing with arbitrary lists. The computer will perform the same operation in each case, but we as programmers (and readers of programs) will be less confused. Another advantage of defining specific functions like `first-name` is that if we decide to change the representation of names we will only have to change the definition of `first-name`. This is a much easier task than hunting through a large program and changing the uses of `first` that refer to names, while leaving other uses alone.

```
(defun first-name (name)
  "Select the first name from a name represented as a list."
  (first name))

> p => (JOHN Q PUBLIC)
> (first-name p) => JOHN
> (first-name '(Wilma Flintstone)) => WILMA
```

```
> (setf names '((John Q Public) (Malcolm X)
                   (Admiral Grace Murray Hopper) (Spot)
                   (Aristotle) (A A Milne) (Z Z Top)
                   (Sir Larry Olivier) (Miss Scarlet))) ⇒
((JOHN Q PUBLIC) (MALCOLM X) (ADMIRAL GRACE MURRAY HOPPER)
 (SPOT) (ARISTOTLE) (A A MILNE) (Z Z TOP) (SIR LARRY OLIVIER)
 (MISS SCARLET))

> (first-name (first names)) ⇒ JOHN
```

In the last expression we used the function `first` to pick out the first element in a list of names, and then the function `first-name` to pick out the first name of that element. We could also have said `(first (first names))` or even `(first (first-name names))` and still have gotten JOHN, but we would not be accurately representing what is being considered a name and what is being considered a list of names.

## 1.6 Using Functions

One good thing about defining a list of names, as we did above, is that it makes it easier to test our functions. Consider the following expression, which can be used to test the `last-name` function:

```
> (mapcar #'last-name names)
(PUBLIC X HOPPER SPOT ARISTOTLE MILNE TOP OLIVIER SCARLET)
```

The funny `#'` notation maps from the name of a function to the function itself. This is analogous to `'x` notation. The built-in function `mapcar` is passed two arguments, a function and a list. It returns a list built by calling the function on every element of the input list. In other words, the `mapcar` call above is equivalent to:

```
(list (last-name (first names))
      (last-name (second names))
      (last-name (third names))
      ...)
```

`mapcar`'s name comes from the fact that it “maps” the function across each of the arguments. The `car` part of the name refers to the Lisp function `car`, an old name for `first`. `cdr` is the old name for `rest`. The names stand for “contents of the address register” and “contents of the decrement register,” the instructions that were used in the first implementation of Lisp on the IBM 704. I'm sure you'll agree that `first` and

rest are much better names, and they will be used instead of car and cdr whenever we are talking about lists. However, we will continue to use car and cdr on occasion when we are considering a pair of values that are not considered as a list. Beware that some programmers still use car and cdr for lists as well.

Here are some more examples of mapcar:

```
> (mapcar #'- '(1 2 3 4)) => (-1 -2 -3 -4)
> (mapcar #'* '(1 2 3 4) '(10 20 30 40)) => (11 22 33 44)
```

This last example shows that mapcar can be passed three arguments, in which case the first argument should be a binary function, which will be applied to corresponding elements of the other two lists. In general, mapcar expects an  $n$ -ary function as its first argument, followed by  $n$  lists. It first applies the function to the argument list obtained by collecting the first element of each list. Then it applies the function to the second element of each list, and so on, until one of the lists is exhausted. It returns a list of all the function values it has computed.

Now that we understand mapcar, let's use it to test the first-name function:

```
> (mapcar #'first-name names)
(JOHN MALCOLM ADMIRAL SPOT ARISTOTLE A Z SIR MISS)
```

We might be disappointed with these results. Suppose we wanted a version of first-name which ignored titles like Admiral and Miss, and got to the "real" first name. We could proceed as follows:

```
(defparameter *titles*
  '(Mr Mrs Miss Ms Sir Madam Dr Admiral Major General)
  "A list of titles that can appear at the start of a name.")
```

We've introduced another new special form, defparameter, which defines a parameter—a variable that does not change over the course of a computation, but that might change when we think of new things to add (like the French Mme or the military Lt.). The defparameter form both gives a value to the variable and makes it possible to use the variable in subsequent function definitions. In this example we have exercised the option of providing a documentation string that describes the variable. It is a widely used convention among Lisp programmers to mark special variables by spelling their names with asterisks on either end. This is just a convention; in Lisp, the asterisk is just another character that has no particular meaning.

We next give a new definition for first-name, which supersedes the previous definition.<sup>5</sup> This definition says that if the first word of the name is a member of the

---

<sup>5</sup>Just as we can change the value of a variable, we can also change the value of a function

list of titles, then we want to ignore that word and return the first-name of the rest of the words in the name. Otherwise, we use the first word, just as before. Another built-in function, `member`, tests to see if its first argument is an element of the list passed as the second argument.

The special form `if` has the form (`if test then-part else-part`). There are many special forms for performing conditional tests in Lisp; `if` is the most appropriate for this example. An `if` form is evaluated by first evaluating the `test` expression. If it is true, the `then-part` is evaluated and returned as the value of the `if` form; otherwise the `else-part` is evaluated and returned. While some languages insist that the value of a conditional test must be either `true` or `false`, Lisp is much more forgiving. The test may legally evaluate to any value at all. Only the value `nil` is considered false; all other values are considered true. In the definition of `first-name` below, the function `member` will return a non-`nil` (hence true) value if the first element of the name is in the list of titles, and will return `nil` (hence false) if it is not. Although all non-`nil` values are considered true, by convention the constant `t` is usually used to represent truth.

```
(defun first-name (name)
  "Select the first name from a name represented as a list."
  (if (member (first name) *titles*)
      (first-name (rest name))
      (first name)))
```

When we map the new `first-name` over the list of names, the results are more encouraging. In addition, the function gets the “right” result for ‘(Madam Major General Paula Jones) by dropping off titles one at a time.

```
> (mapcar #'first-name names)
(JOHN MALCOLM GRACE SPOT ARISTOTLE A Z LARRY SCARLET)

> (first-name '(Madam Major General Paula Jones))
PAULA
```

We can see how this works by *tracing* the execution of `first-name`, and seeing the values passed to and returned from the function. The special forms `trace` and `untrace` are used for this purpose.

```
> (trace first-name)
(FIRST-NAME)
```

---

in Lisp. It is not necessary to recompile everything when a change is made, as it would be in other languages.

```
> (first-name '(John Q Public))
(1 ENTER FIRST-NAME: (JOHN Q PUBLIC))
(1 EXIT FIRST-NAME: JOHN)
JOHN
```

When `first-name` is called, the definition is entered with the single argument, `name`, taking on the value `(JOHN Q PUBLIC)`. The value returned is `JOHN`. Trace prints two lines indicating entry and exit from the function, and then Lisp, as usual, prints the final result, `JOHN`.

The next example is more complicated. The function `first-name` is used four times. First, it is entered with `name` bound to `(Madam Major General Paula Jones)`. The first element of this list is `Madam`, and since this is a member of the list of titles, the result is computed by calling `first-name` again on the rest of the name—`(Major General Paula Jones)`. This process repeats two more times, and we finally enter `first-name` with `name` bound to `(Paula Jones)`. Since `Paula` is not a title, it becomes the result of this call to `first-name`, and thus the result of all four calls, as trace shows. Once we are happy with the workings of `first-name`, the special form `untrace` turns off tracing.

```
> (first-name '(Madam Major General Paula Jones)) =>
(1 ENTER FIRST-NAME: (MADAM MAJOR GENERAL PAULA JONES))
(2 ENTER FIRST-NAME: (MAJOR GENERAL PAULA JONES))
(3 ENTER FIRST-NAME: (GENERAL PAULA JONES))
(4 ENTER FIRST-NAME: (PAULA JONES))
(4 EXIT FIRST-NAME: PAULA)
(3 EXIT FIRST-NAME: PAULA)
(2 EXIT FIRST-NAME: PAULA)
(1 EXIT FIRST-NAME: PAULA)
PAULA
> (untrace first-name) => (FIRST-NAME)
> (first-name '(Mr Blue Jeans)) => BLUE
```

The function `first-name` is said to be *recursive* because its definition includes a call to itself. Programmers who are new to the concept of recursion sometimes find it mysterious. But recursive functions are really no different from nonrecursive ones. Any function is required to return the correct value for the given input(s). Another way to look at this requirement is to break it into two parts: a function must return a value, and it must not return any incorrect values. This two-part requirement is equivalent to the first one, but it makes it easier to think about and design function definitions.

Next I show an abstract description of the `first-name` problem, to emphasize the design of the function and the fact that recursive solutions are not tied to Lisp in any way:

```
function first-name(name):
    if the first element of name is a title
        then do something complicated to get the first-name
    else return the first element of the name
```

This breaks up the problem into two cases. In the second case, we return an answer, and it is in fact the correct answer. We have not yet specified what to do in the first case. But we do know that it has something to do with the rest of the name after the first element, and that what we want is to extract the first name out of those elements. The leap of faith is to go ahead and use `first-name`, even though it has not been fully defined yet:

```
function first-name(name):
    if the first element of name is a title
        then return the first-name of the rest of the name
    else return the first element of the name
```

Now the first case in `first-name` is recursive, and the second case remains unchanged. We already agreed that the second case returns the correct answer, and the first case only returns what `first-name` returns. So `first-name` as a whole can only return correct answers. Thus, we're halfway to showing that the function is correct; the other half is to show that it eventually returns some answer. But every recursive call chops off the first element and looks at the rest, so for an  $n$ -element list there can be at most  $n$  recursive calls. This completes the demonstration that the function is correct. Programmers who learn to think this way find recursion to be a valuable tool rather than a confusing mystery.

## 1.7 Higher-Order Functions

Functions in Lisp can not only be “called,” or applied to arguments, they can also be manipulated just like any other kind of object. A function that takes another function as an argument is called a *higher-order function*. `mapcar` is an example. To demonstrate the higher-order-function style of programming, we will define a new function called `mappend`. It takes two arguments, a function and a list. `mappend` maps the function over each element of the list and appends together all the results. The first definition follows immediately from the description and the fact that the function `apply` can be used to apply a function to a list of arguments.

---

```
(defun mappend (fn the-list)
  "Apply fn to each element of list and append the results."
  (apply #'append (mapcar fn the-list)))
```

Now we experiment a little to see how `apply` and `mappend` work. The first example applies the addition function to a list of four numbers.

```
> (apply #'+ '(1 2 3 4)) ⇒ 10
```

The next example applies `append` to a list of two arguments, where each argument is a list. If the arguments were not lists, it would be an error.

```
> (apply #'append '((1 2 3) (a b c))) ⇒ (1 2 3 A B C)
```

Now we define a new function, `self-and-double`, and apply it to a variety of arguments.

```
> (defun self-and-double (x) (list x (+ x x)))
> (self-and-double 3) ⇒ (3 6)
> (apply #'self-and-double '(3)) ⇒ (3 6)
```

If we had tried to apply `self-and-double` to a list of more than one argument, or to a list that did not contain a number, it would be an error, just as it would be an error to evaluate `(self-and-double 3 4)` or `(self-and-double 'Kim)`. Now let's return to the mapping functions:

```
> (mapcar #'self-and-double '(1 10 300)) ⇒ ((1 2) (10 20) (300 600))
> (mappend #'self-and-double '(1 10 300)) ⇒ (1 2 10 20 300 600)
```

When `mapcar` is passed a function and a list of three arguments, it always returns a list of three values. Each value is the result of calling the function on the respective argument. In contrast, when `mappend` is called, it returns one big list, which is equal to all the values that `mapcar` would generate appended together. It would be an error to call `mappend` with a function that didn't return lists, because `append` expects to see lists as its arguments.

Now consider the following problem: given a list of elements, return a list consisting of all the numbers in the original list and the negation of those numbers. For example, given the list `(testing 1 2 3 test)`, return `(1 -1 2 -2 3 -3)`. This problem can be solved very easily using `mappend` as a component:

```
(defun numbers-and-negations (input)
  "Given a list, return only the numbers and their negations."
  (mappend #'number-and-negation input))

(defun number-and-negation (x)
  "If x is a number, return a list of x and -x."
  (if (numberp x)
      (list x (- x))
      nil))

> (numbers-and-negations '(testing 1 2 3 test)) => (1 -1 2 -2 3 -3)
```

The alternate definition of `mappend` shown in the following doesn't make use of `mapcar`; instead it builds up the list one element at a time:

```
(defun mappend (fn the-list)
  "Apply fn to each element of list and append the results."
  (if (null the-list)
      nil
      (append (funcall fn (first the-list))
              (mappend fn (rest the-list)))))
```

`funcall` is similar to `apply`; it too takes a function as its first argument and applies the function to a list of arguments, but in the case of `funcall`, the arguments are listed separately:

```
> (funcall #'+ 2 3) => 5
> (apply #'+ '(2 3)) => 5
> (funcall #'+ '(2 3)) => Error: (2 3) is not a number.
```

These are equivalent to `(+ 2 3)`, `(+ 2 3)`, and `(+ '(2 3))`, respectively.

So far, every function we have used has been either predefined in Common Lisp or introduced with a `defun`, which pairs a function with a name. It is also possible to introduce a function without giving it a name, using the special syntax `lambda`.

The name *lambda* comes from the mathematician Alonzo Church's notation for functions (Church 1941). Lisp usually prefers expressive names over terse Greek letters, but `lambda` is an exception. A better name would be `make-function`. Lambda derives from the notation in Russell and Whitehead's *Principia Mathematica*, which used a caret over bound variables:  $\hat{x}(x + x)$ . Church wanted a one-dimensional string, so he moved the caret in front:  $\hat{x}(x + x)$ . The caret looked funny with nothing below it, so Church switched to the closest thing, an uppercase lambda,  $\Lambda x(x + x)$ . The  $\Lambda$  was easily confused with other symbols, so eventually the lowercase lambda was substituted:  $\lambda x(x + x)$ . John McCarthy was a student of Church's at Princeton, so when McCarthy invented Lisp in 1958, he adopted the lambda notation. There

were no Greek letters on the keypunches of that era, so McCarthy used `(lambda (x) (+ x x))`, and it has survived to this day. In general, the form of a lambda expression is

`(lambda (parameters...) body...)`

A lambda expression is just a nonatomic *name* for a function, just as `append` is an atomic name for a built-in function. As such, it is appropriate for use in the first position of a function call, but if we want to get at the actual function, rather than its name, we still have to use the `#'` notation. For example:

```
> ((lambda (x) (+ x 2)) 4) => 6
> (funcall #'(lambda (x) (+ x 2)) 4) => 6
```

To understand the distinction we have to be clear on how expressions are evaluated in Lisp. The normal rule for evaluation states that symbols are evaluated by looking up the value of the variable that the symbol refers to. So the `x` in `(+ x 2)` is evaluated by looking up the value of the variable named `x`. A list is evaluated in one of two ways. If the first element of the list is a special form operator, then the list is evaluated according to the syntax rule for that special form. Otherwise, the list represents a function call. The first element is evaluated in a unique way, as a function. This means it can either be a symbol or a lambda expression. In either case, the function named by the first element is applied to the values of the remaining elements in the list. These values are determined by the normal evaluation rules. If we want to refer to a function in a position other than the first element of a function call, we have to use the `#'` notation. Otherwise, the expressions will be evaluated by the normal evaluation rule, and will not be treated as functions. For example:

```
> append => Error: APPEND is not a bound variable
> (lambda (x) (+ x 2)) => Error: LAMBDA is not a function
```

Here are some more examples of the correct use of functions:

```
> (mapcar #'(lambda (x) (+ x x))
           '(1 2 3 4 5)) =>
(2 4 6 8 10)

> (mappend #'(lambda (l) (list l (reverse l)))
           '((1 2 3) (a b c))) =>
((1 2 3) (3 2 1) (A B C) (C B A))
```

Programmers who are used to other languages sometimes fail to see the point of lambda expressions. There are two reasons why lambda expressions are very useful.

First, it can be messy to clutter up a program with superfluous names. Just as it is clearer to write  $(a+b)*(c+d)$  rather than to invent variable names like `temp1` and `temp2` to hold  $a+b$  and  $c+d$ , so it can be clearer to define a function as a lambda expression rather than inventing a name for it.

Second, and more importantly, lambda expressions make it possible to create new functions at run time. This is a powerful technique that is not possible in most programming languages. These run-time functions, known as *closures*, will be covered in section 3.16.

## 1.8 Other Data Types

So far we have seen just four kinds of Lisp objects: numbers, symbols, lists, and functions. Lisp actually defines about 25 different types of objects: vectors, arrays, structures, characters, streams, hash tables, and others. At this point we will introduce one more, the string. As you can see in the following, strings, like numbers, evaluate to themselves. Strings are used mainly for printing out messages, while symbols are used for their relationships to other objects, and to name variables. The printed representation of a string has a double quote mark ("") at each end.

```
> "a string" ⇒ "a string"  
> (length "a string") ⇒ 8  
> (length "") ⇒ 0
```

## 1.9 Summary: The Lisp Evaluation Rule

We can now summarize the evaluation rule for Lisp.

- Every expression is either a *list* or an *atom*.
- Every list to be evaluated is either a *special form expression* or a *function application*.
- A *special form expression* is defined to be a list whose first element is a special form operator. The expression is evaluated according to the operator's idiosyncratic evaluation rule. For example, the evaluation rule for `setf` is to evaluate the second argument according to the normal evaluation rule, set the first argument to that value, and return the value as the result. The rule for `defun` is to define a new function, and return the name of the function. The rule for `quote` is to return the first argument unevaluated. The notation '`x`' is actually an

abbreviation for the special form expression (quote  $x$ ). Similarly, the notation #' $f$  is an abbreviation for the special form expression (function  $f$ ).

```
'John ≡ (quote John) ⇒ JOHN
(setf p 'John) ⇒ JOHN
(defun twice (x) (+ x x)) ⇒ TWICE
(if (= 2 3) (error) (+ 5 6)) ⇒ 11
```

- A *function application* is evaluated by first evaluating the arguments (the rest of the list) and then finding the function named by the first element of the list and applying it to the list of evaluated arguments.

```
(+ 2 3) ⇒ 5
(- (+ 90 9) (+ 50 5 (length '(Pat Kim)))) ⇒ 42
```

Note that if '(Pat Kim) did not have the quote, it would be treated as a function application of the function pat to the value of the variable kim.

- Every atom is either a *symbol* or a *nonsymbol*.
- A *symbol* evaluates to the most recent value that has been assigned to the variable named by that symbol. Symbols are composed of letters, and possibly digits and, rarely, punctuation characters. To avoid confusion, we will use symbols composed mostly of the letters a-z and the '-' character, with a few exceptions.<sup>6</sup>

```
names
p
*print-pretty*
```

- A *nonsymbol atom* evaluates to itself. For now, numbers and strings are the only such non-symbol atoms we know of. Numbers are composed of digits, and possibly a decimal point and sign. There are also provisions for scientific notation, rational and complex numbers, and numbers with different bases, but we won't describe the details here. Strings are delimited by double quote marks on both sides.

---

<sup>6</sup>For example, symbols that denote so-called *special variables* usually begin and end in asterisks. Also, note that I did not hesitate to use the symbol won! on page 11.

```
42  ⇒  42  
-273.15 ⇒ -273.15  
"a string" ⇒ "a string"
```

There are some minor details of Common Lisp that complicate the evaluation rules, but this definition will suffice for now.

One complication that causes confusion for beginning Lispers is the difference between *reading* and *evaluating* an expression. Beginners often imagine that when they type an expression, such as

```
> (+ (* 3 4) (* 5 6))
```

the Lisp system first reads the `(+`, then fetches the addition function, then reads `(* 3 4)` and computes 12, then reads `(* 5 6)` and computes 30, and finally computes 42. In fact, what actually happens is that the system first reads the entire expression, the list `(+ (* 3 4) (* 5 6))`. Only after it has been read does the system begin to evaluate it. This evaluation can be done by an interpreter that looks at the list directly, or it can be done by a compiler that translates the list into machine language instructions and then executes those instructions.

We can see now that it was a little imprecise to say, “Numbers are composed of digits, and possibly a decimal point and sign.” It would be more precise to say that the printed representation of a number, as expected by the function `read` and as produced by the function `print`, is composed of digits, and possibly a decimal point and sign. The internal representation of a number varies from one computer to another, but you can be sure that it will be a bit pattern in a particular memory location, and it will no longer contain the original characters used to represent the number in decimal notation. Similarly, it is the printed representation of a string that is surrounded by double quote marks; the internal representation is a memory location marking the beginning of a vector of characters.

Beginners who fail to grasp the distinction between reading and evaluating may have a good model of what expressions evaluate to, but they usually have a terrible model of the efficiency of evaluating expressions. One student used only one-letter variable names, because he felt that it would be faster for the computer to look up a one-letter name than a multiletter name. While it may be true that shorter names can save a microsecond at read time, this makes no difference at all at evaluation time. Every variable, regardless of its name, is just a memory location, and the time to access the location does not depend on the name of the variable.

## 1.10 What Makes Lisp Different?

What is it that sets Lisp apart from other languages? Why is it a good language for AI applications? There are at least eight important factors:

- Built-in Support for Lists
- Automatic Storage Management
- Dynamic Typing
- First-Class Functions
- Uniform Syntax
- Interactive Environment
- Extensibility
- History

In sum, these factors allow a programmer to delay making decisions. In the example dealing with names, we were able to use the built-in list functions to construct and manipulate names without making a lot of explicit decisions about their representation. If we decided to change the representation, it would be easy to go back and alter parts of the program, leaving other parts unchanged.

This ability to delay decisions—or more accurately, to make temporary, nonbinding decisions—is usually a good thing, because it means that irrelevant details can be ignored. There are also some negative points of delaying decisions. First, the less we tell the compiler, the greater the chance that it may have to produce inefficient code. Second, the less we tell the compiler, the less chance it has of noticing inconsistencies and warning us. Errors may not be detected until the program is run. Let's consider each factor in more depth, weighing the advantages and disadvantages:

- *Built-in Support for Lists.* The list is a very versatile data structure, and while lists can be implemented in any language, Lisp makes it easy to use them. Many AI applications involve lists of constantly changing size, making fixed-length data structures like vectors harder to use.

Early versions of Lisp used lists as their only aggregate data structure. Common Lisp provides other types as well, because lists are not always the most efficient choice.

- *Automatic Storage Management.* The Lisp programmer needn't keep track of memory allocation; it is all done automatically. This frees the programmer of a lot of effort, and makes it easy to use the functional style of programming. Other

languages present programmers with a choice. Variables can be allocated on the stack, meaning that they are created when a procedure is entered, and disappear when the procedure is done. This is an efficient use of storage, but it rules out functions that return complex values. The other choice is for the programmer to explicitly allocate and free storage. This makes the functional style possible but can lead to errors.

For example, consider the trivial problem of computing the expression  $a \times (b + c)$ , where  $a$ ,  $b$ , and  $c$  are numbers. The code is trivial in any language; here it is in Pascal and in Lisp:

```
/* Pascal */           ::; Lisp
a * (b + c)          (* a (+ b c))
```

The only difference is that Pascal uses infix notation and Lisp uses prefix. Now consider computing  $a \times (b + c)$  when  $a$ ,  $b$ , and  $c$  are matrices. Assume we have procedures for matrix multiplication and addition. In Lisp the form is exactly the same; only the names of the functions are changed. In Pascal we have the choice of approaches mentioned before. We could declare temporary variables to hold intermediate results on the stack, and replace the functional expression with a series of procedure calls:

```
/* Pascal */           ::; Lisp
var temp, result: matrix;

add(b,c,temp);          (mult a (add b c))
mult(a,temp,result);
return(result);
```

The other choice is to write Pascal functions that allocate new matrices on the heap. Then one can write nice functional expressions like `mult(a,add(b,c))` even in Pascal. However, in practice it rarely works this nicely, because of the need to manage storage explicitly:

```
/* Pascal */           ::; Lisp
var a,b,c,x,y: matrix;
```

```
x := add(b,c);                                (mult a (add b c))
y := mult(a,x);
free(x);
return(y);
```

In general, deciding which structures to free is a difficult task for the Pascal programmer. If the programmer misses some, then the program may run out of memory. Worse, if the programmer frees a structure that is still being used, then strange errors can occur when that piece of memory is reallocated. Lisp automatically allocates and frees structures, so these two types of errors can *never* occur.

- *Dynamic Typing.* Lisp programmers don't have to provide type declarations, because the language keeps track of the type of each object at run time, rather than figuring out all types at compile time. This makes Lisp programs shorter and hence faster to develop, and it also means that functions can often be extended to work for objects to which they were not originally intended to apply. In Pascal, we can write a procedure to sort an array of 100 integers, but we can't use that same procedure to sort 200 integers, or 100 strings. In Lisp, one sort fits all.

One way to appreciate this kind of flexibility is to see how hard it is to achieve in other languages. It is impossible in Pascal; in fact, the language Modula was invented primarily to fix this problem in Pascal. The language Ada was designed to allow flexible generic functions, and a book by Musser and Stepanov (1989) describes an Ada package that gives some of the functionality of Common Lisp's sequence functions. But the Ada solution is less than ideal: it takes a 264-page book to duplicate only part of the functionality of the 20-page chapter 14 from Steele (1990), and Musser and Stepanov went through five Ada compilers before they found one that would correctly compile their package. Also, their package is considerably less powerful, since it does not handle vectors or optional keyword parameters. In Common Lisp, all this functionality comes for free, and it is easy to add more.

On the other hand, dynamic typing means that some errors will go undetected until run time. The great advantage of strongly typed languages is that they are able to give error messages at compile time. The great frustration with strongly typed languages is that they are only able to warn about a small class of errors. They can tell you that you are mistakenly passing a string to a function that expects an integer, but they can't tell you that you are passing an odd number to a function that expects an even number.

- *First-Class Functions.* A *first-class* object is one that can be used anywhere and can be manipulated in the same ways as any other kind of object. In Pascal or C,

for example, functions can be passed as arguments to other functions, but they are not first-class, because it is not possible to create new functions while the program is running, nor is it possible to create an anonymous function without giving it a name. In Lisp we can do both those things using `lambda`. This is explained in section 3.16, page 92.

- *Uniform Syntax.* The syntax of Lisp programs is simple. This makes the language easy to learn, and very little time is wasted correcting typos. In addition, it is easy to write programs that manipulate other programs or define whole new languages—a very powerful technique. The simple syntax also makes it easy for text editing programs to parse Lisp. Your editor program should be able to indent expressions automatically and to show matching parentheses. This is harder to do for languages with complex syntax.

On the other hand, some people object to all the parentheses. There are two answers to this objection. First, consider the alternative: in a language with “conventional” syntax, Lisp’s parentheses pairs would be replaced either by an implicit operator precedence rule (in the case of arithmetic and logical expressions) or by a begin/end pair (in the case of control structures). But neither of these is necessarily an advantage. Implicit precedence is notoriously error-prone, and begin/end pairs clutter up the page without adding any content. Many languages are moving away from begin/end: C uses { and }, which are equivalent to parentheses, and several modern functional languages (such as Haskell) use horizontal blank space, with no explicit grouping at all.

Second, many Lisp programmers *have* considered the alternative. There have been a number of preprocessors that translate from “conventional” syntax into Lisp. None of these has caught on. It is not that Lisp programmers find it *tolerable* to use all those parentheses, rather, they find it *advantageous*. With a little experience, you may too.

It is also important that the syntax of Lisp data is the same as the syntax of programs. Obviously, this makes it easy to convert data to program. Less obvious is the time saved by having universal functions to handle input and output. The Lisp functions `read` and `print` will automatically handle any list, structure, string, or number. This makes it trivial to test individual functions while developing your program. In a traditional language like C or Pascal, you would have to write special-purpose functions to read and print each data type you wanted to debug, as well as a special-purpose driver to call the routines. Because this is time-consuming and error-prone, the temptation is to avoid testing altogether. Thus, Lisp encourages better-tested programs, and makes it easier to develop them faster.

- *Interactive Environment.* Traditionally, a programmer would write a complete program, compile it, correct any errors detected by the compiler, and then

run and debug it. This is known as the *batch* mode of interaction. For long programs, waiting for the compiler occupied a large portion of the debugging time. In Lisp one normally writes a few small functions at a time, getting feedback from the Lisp system after evaluating each one. This is known as an *interactive* environment. When it comes time to make a change, only the changed functions need to be recompiled, so the wait is much shorter. In addition, the Lisp programmer can debug by typing in arbitrary expressions at any time. This is a big improvement over editing the program to introduce print statements and recompiling.

Notice that the distinction between *interactive* and a *batch* languages is separate from the distinction between *interpreted* and *compiled* languages. It has often been stated, incorrectly, that Lisp has an advantage by virtue of being an interpreted language. Actually, experienced Common Lisp programmers tend to use the compiler almost exclusively. The important point is interaction, not interpretation.

The idea of an interactive environment is such a good one that even traditional languages like C and Pascal are starting to offer interactive versions, so this is not an exclusive advantage of Lisp. However, Lisp still provides much better access to the interactive features. A C interpreter may allow the programmer to type in an expression and have it evaluated immediately, but it will not allow the programmer to write a program that, say, goes through the symbol table and finds all the user-defined functions and prints information on them. In C—even interpreted C—the symbol table is just a Cheshire-cat-like invention of the interpreter’s imagination that disappears when the program is run. In Lisp, the symbol table is a first-class object<sup>7</sup> that can be accessed and modified with functions like `read`, `intern` and `do-symbols`.

Common Lisp offers an unusually rich set of useful tools, including over 700 built-in functions (ANSI Common Lisp has over 900). Thus, writing a new program involves more gathering of existing pieces of code and less writing of new code from scratch. In addition to the standard functions, Common Lisp implementations usually provide extensions for interacting with the editor, debugger, and window system.

- *Extensibility.* When Lisp was invented in 1958, nobody could have foreseen the advances in programming theory and language design that have taken place in the last thirty years. Other early languages have been discarded, replaced by ones based on newer ideas. However, Lisp has been able to survive, because it has been able to adapt. Because Lisp is extensible, it has been changed to incorporate the newest features as they become popular.

---

<sup>7</sup>Actually, there can be several symbol tables. They are known as *packages* in Common Lisp.

The easiest way to extend the language is with macros. When so-called structured programming constructs such as *case* and *if-then-else* arose, they were incorporated into Lisp as macros. But the flexibility of Lisp goes beyond adding individual constructs. Brand new styles of programming can easily be implemented. Many AI applications are based on the idea of *rule-based* programming. Another new style is *object-oriented* programming, which has been incorporated with the Common Lisp Object System (CLOS),<sup>8</sup> a set of macros, functions, and data types that have been integrated into ANSI Common Lisp.

To show how far Lisp has come, here's the only sample program given in the *Lisp/MTS Programmer's Guide* (Hafner and Wilcox 1974):

```
(PROG (LIST DEPTH TEMP RESTLIST)
      (SETQ RESTLIST (LIST (CONS (READ) 0)) )
      A (COND
          ((NOT RESTLIST) (RETURN 'DONE))
          (T (SETQ LIST (UNCONS (UNCONS RESTLIST
                                         RESTLIST ) DEPTH)))
             (COND ((ATOM LIST)
                   (MAPC 'PRIN1 (LIST '"ATOM:" LIST "," 'DEPTH DEPTH))
                   (TERPRI))
                   (T (SETQ TEMP (UNCONS LIST LIST))
                      (COND (LIST
                            (SETQ RESTLIST (CONS(CONS LIST DEPTH) RESTLIST))))
                            (SETQ RESTLIST (CONS (CONS TEMP
                                         (ADD1 DEPTH)) RESTLIST))
                           ))))
             (GO A))
```

Note the use of the now-deprecated *goto* (*GO*) statement, and the lack of consistent indentation conventions. The manual also gives a recursive version of the same program:

```
(PROG NIL (
  (LABEL ATOMPRINT (LAMBDA (RESTLIST)
    (COND ((NOT RESTLIST) (RETURN 'DONE))
          ((ATOM (CAAR RESTLIST)) (MAPC 'PRIN1
            (LIST '"ATOM:" (CAAR RESTLIST)
                  "," 'DEPTH (CDAR RESTLIST)))
            (TERPRI)
            (ATOMPRINT (CDR RESTLIST)))
          ( T (ATOMPRINT (GRAFT
            (LIST (CONS (CAAAR RESTLIST) (ADD1 (CDAR RESTLIST)))))
            (AND (CDAAR RESTLIST) (LIST (CONS (CDAAR RESTLIST)
```

---

<sup>8</sup>Pronounced "see-loss." An alternate pronunciation, "klaus," seems to be losing favor.

```
(CDAR RESTLIST))))  
      (CDR RESTLIST))))))  
(LIST (CONS (READ) 0))))
```

Both versions are very difficult to read. With our modern insight (and text editors that automatically indent), a much simpler program is possible:

```
(defun atomprint (exp &optional (depth 0))  
  "Print each atom in exp, along with its depth of nesting."  
  (if (atom exp)  
      (format t "~&ATOM: ~a, DEPTH ~d" exp depth)  
      (dolist (element exp)  
        (atomprint element (+ depth 1)))))
```

## 1.11 Exercises

-  **Exercise 1.1 [m]** Define a version of `last-name` that handles “Rex Morgan MD,” “Morton Downey, Jr.,” and whatever other cases you can think of.
-  **Exercise 1.2 [m]** Write a function to exponentiate, or raise a number to an integer power. For example: `(power 3 2) = 32 = 9`.
-  **Exercise 1.3 [m]** Write a function that counts the number of atoms in an expression. For example: `(count-atoms '(a (b) c)) = 3`. Notice that there is something of an ambiguity in this: should `(a nil c)` count as three atoms, or as two, because it is equivalent to `(a () c)?`
-  **Exercise 1.4 [m]** Write a function that counts the number of times an expression occurs anywhere within another expression. Example: `(count-anywhere 'a '(a ((a) b) a)) ⇒ 3`.
-  **Exercise 1.5 [m]** Write a function to compute the dot product of two sequences of numbers, represented as lists. The dot product is computed by multiplying corresponding elements and then adding up the resulting products. Example:

`(dot-product '(10 20) '(3 4)) = 10 × 3 + 20 × 4 = 110`

## 1.12 Answers

### Answer 1.2

```
(defun power (x n)
  "Power raises x to the nth power.  N must be an integer >= 0.
  This executes in log n time, because of the check for even n."
  (cond ((= n 0) 1)
        ((evenp n) (expt (power x (/ n 2)) 2))
        (t (* x (power x (- n 1))))))
```

### Answer 1.3

```
(defun count-atoms (exp)
  "Return the total number of non-nil atoms in the expression."
  (cond ((null exp) 0)
        ((atom exp) 1)
        (t (+ (count-atoms (first exp))
               (count-atoms (rest exp))))))

(defun count-all-atoms (exp &optional (if-null 1))
  "Return the total number of atoms in the expression,
  counting nil as an atom only in non-tail position."
  (cond ((null exp) if-null)
        ((atom exp) 1)
        (t (+ (count-all-atoms (first exp) 1)
               (count-all-atoms (rest exp) 0)))))
```

### Answer 1.4

```
(defun count-anywhere (item tree)
  "Count the times item appears anywhere within tree."
  (cond ((eql item tree) 1)
        ((atom tree) 0)
        (t (+ (count-anywhere item (first tree))
               (count-anywhere item (rest tree))))))
```

**Answer 1.5** Here are three versions:

```
(defun dot-product (a b)
  "Compute the mathematical dot product of two vectors."
  (if (or (null a) (null b))
      0
      (+ (* (first a) (first b))
          (dot-product (rest a) (rest b)))))

(defun dot-product (a b)
  "Compute the mathematical dot product of two vectors."
  (let ((sum 0))
    (dotimes (i (length a))
      (incf sum (* (elt a i) (elt b i))))
    sum))

(defun dot-product (a b)
  "Compute the mathematical dot product of two vectors."
  (apply #'+ (mapcar #'* a b)))
```

## CHAPTER 2

---

# A Simple Lisp Program

*Certum quod factum.*  
(One is certain of only what one builds.)  
—Giovanni Battista Vico (1668–1744)  
Italian royal historiographer

**Y**ou will never become proficient in a foreign language by studying vocabulary lists. Rather, you must hear and speak (or read and write) the language to gain proficiency. The same is true for learning computer languages.

This chapter shows how to combine the basic functions and special forms of Lisp into a complete program. If you can learn how to do that, then acquiring the remaining vocabulary of Lisp (as outlined in chapter 3) will be easy.

## 2.1 A Grammar for a Subset of English

The program we will develop in this chapter generates random English sentences. Here is a simple grammar for a tiny portion of English:

```

Sentence ⇒ Noun-Phrase + Verb-Phrase
Noun-Phrase ⇒ Article + Noun
Verb-Phrase ⇒ Verb + Noun-Phrase
Article ⇒ the, a, ...
Noun ⇒ man, ball, woman, table ...
Verb ⇒ hit, took, saw, liked ...

```

To be technical, this description is called a *context-free phrase-structure grammar*, and the underlying paradigm is called *generative syntax*. The idea is that anywhere we want a sentence, we can generate a noun phrase followed by a verb phrase. Anywhere a noun phrase has been specified, we generate instead an article followed by a noun. Anywhere an article has been specified, we generate either “the,” “a,” or some other article. The formalism is “context-free” because the rules apply anywhere regardless of the surrounding words, and the approach is “generative” because the rules as a whole define the complete set of sentences in a language (and by contrast the set of nonsentences as well). In the following we show the derivation of a single sentence using the rules:

```

To get a Sentence, append a Noun-Phrase and a Verb-Phrase
To get a Noun-Phrase, append an Article and a Noun
    Choose “the” for the Article
    Choose “man” for the Noun
The resulting Noun-Phrase is “the man”
To get a Verb-Phrase, append a Verb and a Noun-Phrase
    Choose “hit” for the Verb
To get a Noun-Phrase, append an Article and a Noun
    Choose “the” for the Article
    Choose “ball” for the Noun
The resulting Noun-Phrase is “the ball”
The resulting Verb-Phrase is “hit the ball”
The resulting Sentence is “The man hit the ball”

```

## 2.2 A Straightforward Solution

We will develop a program that generates random sentences from a phrase-structure grammar. The most straightforward approach is to represent each grammar rule by a separate Lisp function:

```
(defun sentence ()    (append (noun-phrase) (verb-phrase)))
(defun noun-phrase () (append (Article) (Noun)))
(defun verb-phrase () (append (Verb) (noun-phrase)))
(defun Article ()    (one-of '(the a)))
(defun Noun ()        (one-of '(man ball woman table)))
(defun Verb ()        (one-of '(hit took saw liked)))
```

Each of these function definitions has an empty parameter list, (). That means the functions take no arguments. This is unusual because, strictly speaking, a function with no arguments would always return the same thing, so we would use a constant instead. However, these functions make use of the random function (as we will see shortly), and thus can return different results even with no arguments. Thus, they are not functions in the mathematical sense, but they are still called functions in Lisp, because they return a value.

All that remains now is to define the function one-of. It takes a list of possible choices as an argument, chooses one of these at random, and returns a one-element list of the element chosen. This last part is so that all functions in the grammar will return a list of words. That way, we can freely apply append to any category.

```
(defun one-of (set)
  "Pick one element of set, and make a list of it."
  (list (random-elt set)))

(defun random-elt (choices)
  "Choose an element from a list at random."
  (elt choices (random (length choices))))
```

There are two new functions here, elt and random. elt picks an element out of a list. The first argument is the list, and the second is the position in the list. The confusing part is that the positions start at 0, so (elt choices 0) is the first element of the list, and (elt choices 1) is the second. Think of the position numbers as telling you how far away you are from the front. The expression (random n) returns an integer from 0 to n-1, so that (random 4) would return either 0,1,2, or 3.

Now we can test the program by generating a few random sentences, along with a noun phrase and a verb phrase:

```
> (sentence) => (THE WOMAN HIT THE BALL)
> (sentence) => (THE WOMAN HIT THE MAN)
> (sentence) => (THE BALL SAW THE WOMAN)
> (sentence) => (THE BALL SAW THE TABLE)
> (noun-phrase) => (THE MAN)
> (verb-phrase) => (LIKED THE WOMAN)
```

```

> (trace sentence noun-phrase verb-phrase article noun verb) ⇒
(SENTENCE NOUN-PHRASE VERB-PHRASE ARTICLE NOUN VERB)

> (sentence) ⇒
(1 ENTER SENTENCE)
(1 ENTER NOUN-PHRASE)
(1 ENTER ARTICLE)
(1 EXIT ARTICLE: (THE))
(1 ENTER NOUN)
(1 EXIT NOUN: (MAN))
(1 EXIT NOUN-PHRASE: (THE MAN))
(1 ENTER VERB-PHRASE)
(1 ENTER VERB)
(1 EXIT VERB: (HIT))
(1 ENTER NOUN-PHRASE)
(1 ENTER ARTICLE)
(1 EXIT ARTICLE: (THE))
(1 ENTER NOUN)
(1 EXIT NOUN: (BALL))
(1 EXIT NOUN-PHRASE: (THE BALL))
(1 EXIT VERB-PHRASE: (HIT THE BALL))
(1 EXIT SENTENCE: (THE MAN HIT THE BALL))
(THE MAN HIT THE BALL)

```

The program works fine, and the trace looks just like the sample derivation above, but the Lisp definitions are a bit harder to read than the original grammar rules. This problem will be compounded as we consider more complex rules. Suppose we wanted to allow noun phrases to be modified by an indefinite number of adjectives and an indefinite number of prepositional phrases. In grammatical notation, we might have the following rules:

$$\begin{aligned}
Noun-Phrase &\Rightarrow Article + Adj^* + Noun + PP^* \\
Adj^* &\Rightarrow \emptyset, Adj + Adj^* \\
PP^* &\Rightarrow \emptyset, PP + PP^* \\
PP &\Rightarrow Prep + Noun-Phrase \\
Adj &\Rightarrow big, little, blue, green, \dots \\
Prep &\Rightarrow to, in, by, with, \dots
\end{aligned}$$

In this notation,  $\emptyset$  indicates a choice of nothing at all, a comma indicates a choice of several alternatives, and the asterisk is nothing special—as in Lisp, it's just part of the name of a symbol. However, the convention used here is that names ending in an asterisk denote zero or more repetitions of the underlying name. That is,  $PP^*$  denotes zero or more repetitions of  $PP$ . This is known as “Kleene star” notation (pronounced

"clean-E") after the mathematician Stephen Cole Kleene.<sup>1</sup>

The problem is that the rules for *Adj\** and *PP\** contain choices that we would have to represent as some kind of conditional in Lisp. For example:

```
(defun Adj* ()
  (if (= (random 2) 0)
      nil
      (append (Adj) (Adj*))))

(defun PP* ()
  (if (random-elt '(t nil))
      (append (PP) (PP*))
      nil))

(defun noun-phrase () (append (Article) (Adj*) (Noun) (PP*)))
(defun PP () (append (Prep) (noun-phrase)))
(defun Adj () (one-of '(big little blue green adiabatic)))
(defun Prep () (one-of '(to in by with on)))
```

I've chosen two different implementations for *Adj\** and *PP\**; either approach would work in either function. We have to be careful, though; here are two approaches that would not work:

```
(defun Adj* ()
  "Warning - incorrect definition of Adjectives."
  (one-of '(nil (append (Adj) (Adj*)))))

(defun Adj* ()
  "Warning - incorrect definition of Adjectives."
  (one-of (list nil (append (Adj) (Adj*)))))
```

The first definition is wrong because it could return the literal expression ((append (Adj) (Adj\*))) rather than a list of words as expected. The second definition would cause infinite recursion, because computing the value of (Adj\*) always involves a recursive call to (Adj\*). The point is that what started out as simple functions are now becoming quite complex. To understand them, we need to know many Lisp conventions—`defun`, `()`, `case`, `if`, `quote`, and the rules for order of evaluation—when ideally the implementation of a grammar rule should use only *linguistic* conventions. If we wanted to develop a larger grammar, the problem could get worse, because the rule-writer might have to depend more and more on Lisp.

---

<sup>1</sup>We will soon see "Kleene plus" notation, wherein *PP+* denotes one or more repetition of *PP*.

## 2.3 A Rule-Based Solution

An alternative implementation of this program would concentrate on making it easy to write grammar rules and would worry later about how they will be processed. Let's look again at the original grammar rules:

```

Sentence ⇒ Noun-Phrase + Verb-Phrase
Noun-Phrase ⇒ Article + Noun
Verb-Phrase ⇒ Verb + Noun-Phrase
Article ⇒ the, a, ...
Noun ⇒ man, ball, woman, table ...
Verb ⇒ hit, took, saw, liked ...

```

Each rule consists of an arrow with a symbol on the left-hand side and something on the right-hand side. The complication is that there can be two kinds of right-hand sides: a concatenated list of symbols, as in “*Noun-Phrase* ⇒ *Article + Noun*,” or a list of alternate words, as in “*Noun* ⇒ *man, ball, ...*” We can account for these possibilities by deciding that every rule will have a list of possibilities on the right-hand side, and that a concatenated list, *for example* “*Article + Noun*,” will be represented as a Lisp list, *for example* “(*Article Noun*)”. The list of rules can then be represented as follows:

```

(defparameter *simple-grammar*
  '((sentence -> (noun-phrase verb-phrase))
    (noun-phrase -> (Article Noun))
    (verb-phrase -> (Verb noun-phrase))
    (Article -> the a)
    (Noun -> man ball woman table)
    (Verb -> hit took saw liked))
  "A grammar for a trivial subset of English.")

(defvar *grammar* *simple-grammar*
  "The grammar used by generate. Initially, this is
  *simple-grammar*, but we can switch to other grammars.")

```

Note that the Lisp version of the rules closely mimics the original version. In particular, I include the symbol “*->*”, even though it serves no real purpose; it is purely decorative.

The special forms `defvar` and `defparameter` both introduce special variables and assign a value to them; the difference is that a *variable*, like `*grammar*`, is routinely changed during the course of running the program. A *parameter*, like `*simple-grammar*`, on the other hand, will normally stay constant. A change to a parameter is considered a change *to* the program, not a change *by* the program.

Once the list of rules has been defined, it can be used to find the possible rewrites of a given category symbol. The function `assoc` is designed for just this sort of task.

It takes two arguments, a “key” and a list of lists, and returns the first element of the list of lists that starts with the key. If there is none, it returns `nil`. Here is an example:

```
> (assoc 'noun *grammar*) => (NOUN -> MAN BALL WOMAN TABLE)
```

Although rules are quite simply implemented as lists, it is a good idea to impose a layer of abstraction by defining functions to operate on the rules. We will need three functions: one to get the right-hand side of a rule, one for the left-hand side, and one to look up all the possible rewrites (right-hand sides) for a category.

```
(defun rule-lhs (rule)
  "The left-hand side of a rule."
  (first rule))

(defun rule-rhs (rule)
  "The right-hand side of a rule."
  (rest (rest rule)))

(defun rewrites (category)
  "Return a list of the possible rewrites for this category."
  (rule-rhs (assoc category *grammar*)))
```

Defining these functions will make it easier to read the programs that use them, and it also makes changing the representation of rules easier, should we ever decide to do so.

We are now ready to address the main problem: defining a function that will generate sentences (or noun phrases, or any other category). We will call this function `generate`. It will have to contend with three cases: (1) In the simplest case, `generate` is passed a symbol that has a set of rewrite rules associated with it. We choose one of those at random, and then `generate` from that. (2) If the symbol has no possible rewrite rules, it must be a terminal symbol—a word, rather than a grammatical category—and we want to leave it alone. Actually, we return the list of the input word, because, as in the previous program, we want all results to be lists of words. (3) In some cases, when the symbol has rewrites, we will pick one that is a list of symbols, and try to generate from that. Thus, `generate` must also accept a list as input, in which case it should generate each element of the list, and then append them all together. In the following, the first clause in `generate` handles this case, while the second clause handles (1) and the third handles (2). Note that we used the `mappend` function from section 1.7 (page 18).

```
(defun generate (phrase)
  "Generate a random sentence or phrase"
  (cond ((listp phrase)
         (mappend #'generate phrase)))
```

---

```
((rewrites phrase)
  (generate (random-elt (rewrites phrase))))
  (t (list phrase)))
```

Like many of the programs in this book, this function is short, but dense with information: the craft of programming includes knowing what *not* to write, as well as what to write.

This style of programming is called *data-driven* programming, because the data (the list of rewrites associated with a category) drives what the program does next. It is a natural and easy-to-use style in Lisp, leading to concise and extensible programs, because it is always possible to add a new piece of data with a new association without having to modify the original program.

Here are some examples of generate in use:

```
> (generate 'sentence) => (THE TABLE SAW THE BALL)
> (generate 'sentence) => (THE WOMAN HIT A TABLE)
> (generate 'noun-phrase) => (THE MAN)
> (generate 'verb-phrase) => (TOOK A TABLE)
```

There are many possible ways to write generate. The following version uses if instead of cond:

```
(defun generate (phrase)
  "Generate a random sentence or phrase"
  (if (listp phrase)
      (mappend #'generate phrase)
      (let ((choices (rewrites phrase)))
        (if (null choices)
            (list phrase)
            (generate (random-elt choices))))))
```

This version uses the special form let, which introduces a new variable (in this case, choices) and also binds the variable to a value. In this case, introducing the variable saves us from calling the function rewrites twice, as was done in the cond version of generate. The general form of a let form is:

```
(let ((var value)...)
  body-containing-vars)
```

let is the most common way of introducing variables that are not parameters of functions. One must resist the temptation to use a variable without introducing it:

```
(defun generate (phrase)
  (setf choices ...)           ;;= wrong!
  ... choices ...)
```

This is wrong because the symbol `choices` now refers to a special or global variable, one that may be shared or changed by other functions. Thus, the function `generate` is not reliable, because there is no guarantee that `choices` will retain the same value from the time it is set to the time it is referenced again. With `let` we introduce a brand new variable that nobody else can access; therefore it is guaranteed to maintain the proper value.

- ?
- Exercise 2.1 [m]** Write a version of `generate` that uses `cond` but avoids calling `rewrites` twice.
  
- ?
- Exercise 2.2 [m]** Write a version of `generate` that explicitly differentiates between terminal symbols (those with no rewrite rules) and nonterminal symbols.

## 2.4 Two Paths to Follow

The two versions of the preceding program represent two alternate approaches that come up time and time again in developing programs: (1) Use the most straightforward mapping of the problem description directly into Lisp code. (2) Use the most natural notation available to solve the problem, and then worry about writing an interpreter for that notation.

Approach (2) involves an extra step, and thus is more work for small problems. However, programs that use this approach are often easier to modify and expand. This is especially true in a domain where there is a lot of data to account for. The grammar of natural language is one such domain—in fact, most AI problems fit this description. The idea behind approach (2) is to work with the problem as much as possible in its own terms, and to minimize the part of the solution that is written directly in Lisp.

Fortunately, it is very easy in Lisp to design new notations—in effect, new programming languages. Thus, Lisp encourages the construction of more robust programs. Throughout this book, we will be aware of the two approaches. The reader may notice that in most cases, we choose the second.

## 2.5 Changing the Grammar without Changing the Program

We show the utility of approach (2) by defining a new grammar that includes adjectives, prepositional phrases, proper names, and pronouns. We can then apply the generate function without modification to this new grammar.

```
(defparameter *bigger-grammar*
  '((sentence -> (noun-phrase verb-phrase))
    (noun-phrase -> (Article Adj* Noun PP*) (Name) (Pronoun))
    (verb-phrase -> (Verb noun-phrase PP*))
    (PP* -> () (PP PP*))
    (Adj* -> () (Adj Adj*))
    (PP -> (Prep noun-phrase))
    (Prep -> to in by with on)
    (Adj -> big little blue green adiabatic)
    (Article -> the a)
    (Name -> Pat Kim Lee Terry Robin)
    (Noun -> man ball woman table)
    (Verb -> hit took saw liked)
    (Pronoun -> he she it these those that)))
  (setf *grammar* *bigger-grammar*)
  > (generate 'sentence)
  (A TABLE ON A TABLE IN THE BLUE ADIABATIC MAN SAW ROBIN
   WITH A LITTLE WOMAN)
  > (generate 'sentence)
  (TERRY SAW A ADIABATIC TABLE ON THE GREEN BALL BY THAT WITH KIM
   IN THESE BY A GREEN WOMAN BY A LITTLE ADIABATIC TABLE IN ROBIN
   ON LEE)
  > (generate 'sentence)
  (THE GREEN TABLE HIT IT WITH HE))
```

Notice the problem with case agreement for pronouns: the program generated “with he,” although “with him” is the proper grammatical form. Also, it is clear that the program does not distinguish sensible from silly output.

## 2.6 Using the Same Data for Several Programs

Another advantage of representing information in a declarative form—as rules or facts rather than as Lisp functions—is that it can be easier to use the information for multiple purposes. Suppose we wanted a function that would generate not just the

list of words in a sentence but a representation of the complete syntax of a sentence. For example, instead of the list (a woman took a ball), we want to get the nested list:

```
(SENTENCE (NOUN-PHRASE (ARTICLE A) (NOUN WOMAN))
          (VERB-PHRASE (VERB TOOK)
                        (NOUN-PHRASE (ARTICLE A) (NOUN BALL))))
```

This corresponds to the tree that linguists draw as in figure 2.1.

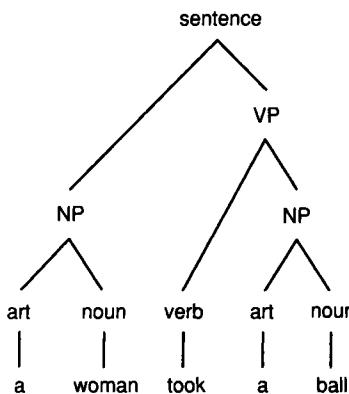


Figure 2.1: Sentence Parse Tree

Using the “straightforward functions” approach we would be stuck; we’d have to rewrite every function to generate the additional structure. With the “new notation” approach we could keep the grammar as it is and just write one new function: a version of generate that produces nested lists. The two changes are to cons the category onto the front of each rewrite, and then not to append together the results but rather just list them with mapcar:

```
(defun generate-tree (phrase)
  "Generate a random sentence or phrase,
  with a complete parse tree."
  (cond ((listp phrase)
         (mapcar #'generate-tree phrase))
        ((rewrites phrase)
         (cons phrase
               (generate-tree (random-elt (rewrites phrase))))))
        (t (list phrase))))
```

Here are some examples:

```

> (generate-tree 'Sentence)
(SENTENCE (NOUN-PHRASE (ARTICLE A)
                         (ADJ*)
                         (NOUN WOMAN)
                         (PP*))
           (VERB-PHRASE (VERB HIT)
                         (NOUN-PHRASE (PRONOUN HE))
                         (PP*)))

> (generate-tree 'Sentence)
(SENTENCE (NOUN-PHRASE (ARTICLE A)
                         (NOUN WOMAN))
           (VERB-PHRASE (VERB TOOK)
                         (NOUN-PHRASE (ARTICLE A) (NOUN BALL))))

```

As another example of the one-data/multiple-program approach, we can develop a function to generate all possible rewrites of a phrase. The function `generate-all` returns a list of phrases rather than just one, and we define an auxiliary function, `combine-all`, to manage the combination of results. Also, there are four cases instead of three, because we have to check for nil explicitly. Still, the complete program is quite simple:

```

(defun generate-all (phrase)
  "Generate a list of all possible expansions of this phrase."
  (cond ((null phrase) (list nil))
        ((listp phrase)
         (combine-all (generate-all (first phrase))
                      (generate-all (rest phrase))))
        ((rewrites phrase)
         (mappend #'(lambda (y)
                      (mapcar #'(lambda (x)
                                  (append x y)) xlist))
                  ylist)))
        (t (list (list phrase)))))

(defun combine-all (xlist ylist)
  "Return a list of lists formed by appending a y to an x.
E.g., (combine-all '((a) (b)) '((1) (2)))
      -> ((A 1) (B 1) (A 2) (B 2))."
  (mappend #'(lambda (y)
               (mapcar #'(lambda (x)
                           (append x y)) xlist))
            ylist))

```

We can now use `generate-all` to test our original little grammar. Note that a serious drawback of `generate-all` is that it can't deal with recursive grammar rules like ' $\text{Adj}^* \Rightarrow \text{Adj} + \text{Adj}^*$ ' that appear in \*bigger-grammar\*, since these lead to an infinite number of outputs. But it works fine for finite languages, like the language generated by \*simple-grammar\*:

```
> (generate-all 'Article)
((THE) (A))

> (generate-all 'Noun)
((MAN) (BALL) (WOMAN) (TABLE))

> (generate-all 'noun-phrase)
((A MAN) (A BALL) (A WOMAN) (A TABLE)
 (THE MAN) (THE BALL) (THE WOMAN) (THE TABLE))

> (length (generate-all 'sentence))
256
```

There are 256 sentences because every sentence in this language has the form Article-Noun-Verb-Article-Noun, and there are two articles, four nouns and four verbs ( $2 \times 4 \times 4 \times 2 \times 4 = 256$ ).

## 2.7 Exercises

-  **Exercise 2.3 [h]** Write a trivial grammar for some other language. This can be a natural language other than English, or perhaps a subset of a computer language.
  
-  **Exercise 2.4 [m]** One way of describing `combine-all` is that it calculates the cross-product of the function `append` on the argument lists. Write the higher-order function `cross-product`, and define `combine-all` in terms of it.  
The moral is to make your code as general as possible, because you never know what you may want to do with it next.

## 2.8 Answers

### Answer 2.1

```
(defun generate (phrase)
  "Generate a random sentence or phrase"
  (let ((choices nil))
    (cond ((listp phrase)
           (mappend #'generate phrase))
          ((setf choices (rewrites phrase))
           (generate (random-elt choices)))
          (t (list phrase)))))
```

**Answer 2.2**

```
(defun generate (phrase)
  "Generate a random sentence or phrase"
  (cond ((listp phrase)
          (mappend #'generate phrase))
        ((non-terminal-p phrase)
          (generate (random-elt (rewrites phrase))))
        (t (list phrase)))))

(defun non-terminal-p (category)
  "True if this is a category in the grammar."
  (not (null (rewrites category))))
```

**Answer 2.4**

```
(defun cross-product (fn xlist ylist)
  "Return a list of all (fn x y) values."
  (mappend #'(lambda (y)
    (mapcar #'(lambda (x) (funcall fn x y))
            xlist))
            ylist))

(defun combine-all (xlist ylist)
  "Return a list of lists formed by appending a y to an x"
  (cross-product #'append xlist ylist))
```

Now we can use the cross-product in other ways as well:

```
> (cross-product #'+ '(1 2 3) '(10 20 30))
(11 12 13
 21 22 23
 31 32 33)

> (cross-product #'list '(a b c d e f g h)
                  '(1 2 3 4 5 6 7 8))
((A 1) (B 1) (C 1) (D 1) (E 1) (F 1) (G 1) (H 1)
 (A 2) (B 2) (C 2) (D 2) (E 2) (F 2) (G 2) (H 2)
 (A 3) (B 3) (C 3) (D 3) (E 3) (F 3) (G 3) (H 3)
 (A 4) (B 4) (C 4) (D 4) (E 4) (F 4) (G 4) (H 4)
 (A 5) (B 5) (C 5) (D 5) (E 5) (F 5) (G 5) (H 5)
 (A 6) (B 6) (C 6) (D 6) (E 6) (F 6) (G 6) (H 6)
 (A 7) (B 7) (C 7) (D 7) (E 7) (F 7) (G 7) (H 7)
 (A 8) (B 8) (C 8) (D 8) (E 8) (F 8) (G 8) (H 8))
```

## CHAPTER 3

---

# Overview of Lisp

*No doubt about it. Common Lisp is a big language.*

—Guy L. Steele, Jr.  
Foreword to Koschman 1990

This chapter briefly covers the most important special forms and functions in Lisp. It can be safely skipped or skimmed by the experienced Common Lisp programmer but is required reading for the novice Lisp programmer, or one who is new to the Common Lisp dialect.

This chapter can be used as a reference source, but the definitive reference is Steele's *Common Lisp the Language*, 2d edition, which should be consulted whenever there is any confusion. Since that book is 25 times longer than this chapter, it is clear that we can only touch on the important highlights here. More detailed coverage is given later in this book as each feature is used in a real program.

## 3.1 A Guide to Lisp Style

The beginning Common Lisp programmer is often overwhelmed by the number of options that the language provides. In this chapter we show fourteen different ways to find the length of a list. How is the programmer to choose between them? One answer is by reading examples of good programs—as illustrated in this book—and copying that style. In general, there are six maxims that every programmer should follow:

- Be specific.
- Use abstractions.
- Be concise.
- Use the provided tools.
- Don't be obscure.
- Be consistent.

These require some explanation.

Using the most specific form possible makes it easier for your reader to understand your intent. For example, the conditional special form `when` is more specific than `if`. The reader who sees a `when` knows to look for only one thing: the clause to consider when the test is true. The reader who sees an `if` can rightfully expect two clauses: one for when the test is true, and one for when it is false. Even though it is possible to use `if` when there is only one clause, it is preferable to use `when`, because `when` is more specific.

One important way of being specific is using abstractions. Lisp provides very general data structures, such as lists and arrays. These can be used to implement specific data structures that your program will use, but you should not make the mistake of invoking primitive functions directly. If you define a list of names:

```
(defvar *names* '((Robert E. Lee) ...))
```

then you should also define functions to get at the components of each name. To get at Lee, use `(last-name (first *names*))`, not `(caddr *names*)`.

Often the maxims are in concord. For example, if your code is trying to find an element in a list, you should use `find` (or maybe `find-if`), not `loop` or `do`. `find` is more specific than the general constructs `loop` or `do`, it is an abstraction, it is more concise, it is a built-in tool, and it is simple to understand.

Sometimes, however, the maxims are in conflict, and experience will tell you which one to prefer. Consider the following two ways of placing a new key/value pair on an association list:<sup>1</sup>

```
(push (cons key val) a-list)
(setf a-list (acons key val a-list))
```

The first is more concise. But the second is more specific, as it uses the `acons` function, which is designed specifically for association lists. The decision between them probably hinges on obscurity: those who find `acons` to be a familiar function would prefer the second, and those who find it obscure would prefer the first.

A similar choice arises in the question of setting a variable to a value. Some prefer `(setq x val)` because it is most specific; others use `(setf x val)`, feeling that it is more consistent to use a single form, `setf`, for all updating. Whichever choice you make on such issues, remember the sixth maxim: be consistent.

## 3.2 Special Forms

As noted in chapter 1, “special form” is the term used to refer both to Common Lisp’s syntactic constructs and the reserved words that mark these constructs. The most commonly used special forms are:

definitions	conditional	variables	iteration	other
<code>defun</code>	<code>and</code>	<code>let</code>	<code>do</code>	<code>declare</code>
<code>defstruct</code>	<code>case</code>	<code>let*</code>	<code>do*</code>	<code>function</code>
<code>defvar</code>	<code>cond</code>	<code>pop</code>	<code>dolist</code>	<code>progn</code>
<code>defparameter</code>	<code>if</code>	<code>push</code>	<code>dotimes</code>	<code>quote</code>
<code>defconstant</code>	<code>or</code>	<code>setf</code>	<code>loop</code>	<code>return</code>
<code>defmacro</code>	<code>unless</code>	<code>incf</code>		<code>trace</code>
<code>labels</code>	<code>when</code>	<code>decf</code>		<code>untrace</code>

To be precise, only `declare`, `function`, `if`, `labels`, `let`, `let*`, `progn` and `quote` are true special forms. The others are actually defined as macros that expand into calls to more primitive special forms and functions. There is no real difference to the programmer, and Common Lisp implementations are free to implement macros as special forms and vice versa, so for simplicity we will continue to use “special form” as a blanket term for both true special forms and built-in macros.

---

<sup>1</sup> Association lists are covered in section 3.6.

### *Special Forms for Definitions*

In this section we survey the special forms that can be used to introduce new global functions, macros, variables, and structures. We have already seen the `defun` form for defining functions; the `defmacro` form is similar and is covered on page 66.

```
(defun function-name (parameter...) "optional documentation" body...)  
(defmacro macro-name (parameter...) "optional documentation" body...)
```

There are three forms for introducing special variables. `defvar` defines a special variable and can optionally be used to supply an initial value and a documentation string. The initial value is evaluated and assigned only if the variable does not yet have any value. `defparameter` is similar, except that the value is required, and it will be used to change any existing value. `defconstant` is used to declare that a symbol will always stand for a particular value.

```
(defvar variable-name initial-value "optional documentation")  
(defparameter variable-name value "optional documentation")  
(defconstant variable-name value "optional documentation")
```

All the `def-` forms define global objects. It is also possible to define local variables with `let`, and to define local functions with `labels`, as we shall see.

Most programming languages provide a way to group related data together into a structure. Common Lisp is no exception. The `defstruct` special form defines a structure type (known as a *record* type in Pascal) and automatically defines functions to get at components of the structure. The general syntax is:

```
(defstruct structure-name "optional documentation" slot...)
```

As an example, we could define a structure for names:

```
(defstruct name  
  first  
  (middle nil)  
  last)
```

This automatically defines the constructor function `make-name`, the recognizer predicate `name-p`, and the accessor functions `name-first`, `name-middle` and `name-last`. The `(middle nil)` means that each new name built by `make-name` will have a middle name of `nil` by default. Here we create, access, and modify a structure:

```

> (setf b (make-name :first 'Barney :last 'Rubble)) ⇒
#S(NAME :FIRST BARNEY :LAST RUBBLE)

> (name-first b) ⇒ BARNEY

> (name-middle b) ⇒ NIL

> (name-last b) ⇒ RUBBLE

> (name-p b) ⇒ T

> (name-p 'Barney) ⇒ NIL      ; only the results of make-name are names

> (setf (name-middle b) 'Q) ⇒ Q

> b ⇒ #S(NAME :FIRST BARNEY :MIDDLE Q :LAST RUBBLE)

```

The printed representation of a structure starts with a #S and is followed by a list consisting of the type of the structure and alternating pairs of slot names and values. Do not let this representation fool you: it is a convenient way of printing the structure, but it is not an accurate picture of the way structures are represented internally. Structures are actually implemented much like vectors. For the name structure, the type would be in the zero element of the vector, the first name in the first element, middle in the second, and last in the third. This means structures are more efficient than lists: they take up less space, and any element can be accessed in a single step. In a list, it takes  $n$  steps to access the  $n$ th element.

There are options that give more control over the structure itself and the individual slots. They will be covered later as they come up.

### *Special Forms for Conditionals*

We have seen the special form `if`, which has the form (`if test then-part else-part`), where either the *then-part* or the *else-part* is the value, depending on the success of the *test*. Remember that only `nil` counts as false; all other values are considered true for the purpose of conditionals. However, the constant `t` is the conventional value used to denote truth (unless there is a good reason for using some other value).

There are actually quite a few special forms for doing conditional evaluation. Technically, `if` is defined as a special form, while the other conditionals are macros, so in some sense `if` is supposed to be the most basic. Some programmers prefer to use `if` for most of their conditionals; others prefer `cond` because it has been around the longest and is versatile (if not particularly pretty). Finally, some programmers opt for a style more like English prose, and freely use `when`, `unless`, `if`, and all the others.

The following table shows how each conditional can be expressed in terms of `if` and `cond`. Actually, these translations are not quite right, because `or`, `case`, and `cond` take care not to evaluate any expression more than once, while the translations with `if` can lead to multiple evaluation of some expressions. The table also has

translations to cond. The syntax of cond is a series of *cond-clauses*, each consisting of a test expression followed by any number of *result* expressions:

```
(cond (test result...)
      (test result...)
      ...)
```

cond goes through the cond-clauses one at a time, evaluating each test expression. As soon as a test expression evaluates non-nil, the result expressions for that clause are each evaluated, and the last expression in the clause is the value of the whole cond. In particular, if a cond-clause consists of just a test and no result expressions, then the value of the cond is the test expression itself, if it is non-nil. If all of the test expressions evaluate to nil, then nil is returned as the value of the cond. A common idiom is to make the last cond-clause be (t result...).

The forms when and unless operate like a single cond clause. Both forms consist of a test followed by any number of consequents, which are evaluated if the test is satisfied—that is, if the test is true for when or false for unless.

The and form tests whether every one of a list of conditions is true, and or tests whether any one is true. Both evaluate the arguments left to right, and stop as soon as the final result can be determined. Here is a table of equivalences:

conditional	if form	cond form
(when test a b c)	(if test (progn a b c))	(cond (test a b c))
(unless test x y)	(if (not test) (progn x y))	(cond ((not test) x y))
(and a b c)	(if a (if b c))	(cond (a (cond (b c))))
(or a b c)	(if a a (if b b c))	(cond (a) (b) (c))
(case a (b c) (t x))	(if (eql a 'b) c x)	(cond ((eql a 'b) c) (t x))

It is considered poor style to use and and or for anything other than testing a logical condition. when, unless, and if can all be used for taking conditional action. For example:

```
(and (> n 100)
     (princ "N is large.")) ; Bad style!
(or (<= n 100)
    (princ "N is large.")) ; Even worse style!
(cond ((> n 100)           ; OK, but not MY preference
       (princ "N is large."))
      (when (> n 100)
          (princ "N is large.")) ; Good style.
```

When the main purpose is to return a value rather than take action, cond and if (with explicit nil in the else case) are preferred over when and unless, which implicitly

return nil in the else case. when and unless are preferred when there is only one possibility, if (or, for some people, cond) when there are two, and cond when there are more than two:

```
(defun tax-bracket (income)
  "Determine what percent tax should be paid for this income."
  (cond ((< income 10000.00) 0.00)
        ((< income 30000.00) 0.20)
        ((< income 50000.00) 0.25)
        ((< income 70000.00) 0.30)
        (t 0.35)))
```

If there are several tests comparing an expression to constants, then case is appropriate. A case form looks like:

```
(case expression
  (match result...)...)
```

The *expression* is evaluated and compared to each successive *match*. As soon as one is eq1, the *result* expressions are evaluated and the last one is returned. Note that the *match* expressions are *not* evaluated. If a *match* expression is a list, then case tests if the *expression* is eq1 to any member of the list. If a *match* expression is the symbol otherwise (or the symbol t), then it matches anything. (It only makes sense for this otherwise clause to be the last one.)

There is also another special form, typecase, which compares the type of an expression against several possibilities and, like case, chooses the first clause that matches. In addition, the special forms ecase and etypecase are just like case and typecase except that they signal an error if there is no match. You can think of the e as standing for either “exhaustive” or “error.” The forms ccase and ctypecase also signal errors, but they can be continuable errors (as opposed to fatal errors): the user is offered the chance to change the expression to something that satisfies one of the matches. Here are some examples of case forms and their cond equivalents:

(case x (1 10) (2 20))	(cond ((eq1 x 1) 10) ((eq1 x 2) 20))
(typecase x (number (abs x)) (list (length x)))	(cond ((typep x 'number) (abs x)) ((typep x 'list) (length x)))
(ecase x (1 10) (2 20))	(cond ((eq1 x 1) 10) ((eq1 x 2) 20) (t (error "no valid case")))

```
(etypecase x
  (number (abs x))
  (list (length x)))
(cond
  ((typep x 'number) (abs x))
  ((typep x 'list) (length x))
  (t (error "no valid typecase")))
```

### *Special Forms for Dealing with Variables and Places*

The special form `setf` is used to assign a new value to a variable or *place*, much as an assignment statement with `=` or `:=` is used in other languages. A place, or *generalized variable* is a name for a location that can have a value stored in it. Here is a table of corresponding assignment forms in Lisp and Pascal:

<code>;; Lisp</code>	<code>/* Pascal */</code>
<code>(setf x 0)</code>	<code>x := 0;</code>
<code>(setf (aref A i j) 0)</code>	<code>A[i,j] := 0;</code>
<code>(setf (rest list) nil)</code>	<code>list^.rest := nil;</code>
<code>(setf (name-middle b) 'Q)</code>	<code>b^.middle := "Q";</code>

`setf` can be used to set a component of a structure as well as to set a variable. In languages like Pascal, the expressions that can appear on the left-hand side of an assignment statement are limited by the syntax of the language. In Lisp, the user can extend the expressions that are allowed in a `setf` form using the special forms `defsetf` or `define-setf-method`. These are introduced on pages 514 and 884 respectively.

There are also some built-in functions that modify places. For example, `(rplacd list nil)` has the same effect as `(setf (rest list) nil)`, except that it returns `list` instead of `nil`. Most Common Lisp programmers prefer to use the `setf` forms rather than the specialized functions.

If you only want to set a variable, the special form `setq` can be used instead. In this book I choose to use `setf` throughout, opting for consistency over specificity.

The discussion in this section makes it seem that variables (and slots of structures) are assigned new values all the time. Actually, many Lisp programs do no assignments whatsoever. It is very common to use Lisp in a functional style where new variables may be introduced, but once a new variable is established, it never changes. One way to introduce a new variable is as a parameter of a function. It is also possible to introduce local variables using the special form `let`. Following are the general `let` form, along with an example. Each variable is bound to the corresponding value, and then the body is evaluated:

```
(let ((variable value)...)
      body...)
      (let ((x 40)
            (y (+ 1 1)))
        (+ x y)) ⇒ 42
```

Defining a local variable with a `let` form is really no different from defining parameters to an anonymous function. The former is equivalent to:

```
((lambda (variable...)
      body...)
      value...)
      ((lambda (x y)
            (+ x y))
       40
       (+ 1 1)))
```

First, all the values are evaluated. Then they are bound to the variables (the parameters of the lambda expression), and finally the body is evaluated, using those bindings.

The special form `let*` is appropriate when you want to use one of the newly introduced variables in a subsequent `value` computation. For example:

```
(let* ((x 6)
      (y (* x x)))
  (+ x y)) ⇒ 42
```

We could not have used `let` here, because then the variable `x` would be unbound during the computation of `y`'s value.



**Exercise 3.1 [m]** Show a `lambda` expression that is equivalent to the above `let*` expression. You may need more than one `lambda`.

Because lists are so important to Lisp, there are special forms for adding and deleting elements from the front of a list—in other words, for treating a list as a stack. If `list` is the name of a location that holds a list, then `(push x list)` will change `list` to have `x` as its first element, and `(pop list)` will return the first element and, as a side-effect, change `list` to no longer contain the first element. `push` and `pop` are equivalent to the following expressions:

```
(push x list) ≡ (setf list (cons x list))
(pop list)   ≡ (let ((result (first list)))
                  (setf list (rest list))
                  result)
```

Just as a list can be used to accumulate elements, a running sum can be used to accumulate numbers. Lisp provides two more special forms, `incf` and `decf`, that can be used to increment or decrement a sum. For both forms the first argument must

be a location (a variable or other setf-able form) and the second argument, which is optional, is the number to increment or decrement by. For those who know C, (incf x) is equivalent to `++x`, and (incf x 2) is equivalent to `x+=2`. In Lisp the equivalence is:

```
(incf x) ≡ (incf x 1) ≡ (setf x (+ x 1))
(decf x) ≡ (decf x 1) ≡ (setf x (- x 1))
```

When the location is a complex form rather than a variable, Lisp is careful to expand into code that does not evaluate any subform more than once. This holds for push, pop, incf, and decf. In the following example, we have a list of players and want to decide which player has the highest score, and thus has won the game. The structure `player` has slots for the player's score and number of wins, and the function `determine-winner` increments the winning player's `wins` field. The expansion of the `incf` form binds a temporary variable so that the sort is not done twice.

```
(defstruct player (score 0) (wins 0))

(defun determine-winner (players)
  "Increment the WINS for the player with highest score."
  (incf (player-wins (first (sort players #'>
                                         :key #'player-score)))))

≡

(defun determine-winner (players)
  "Increment the WINS for the player with highest score."
  (let ((temp (first (sort players #'> :key #'player-score))))
    (setf (player-wins temp) (+ (player-wins temp) 1))))
```

### *Functions and Special Forms for Repetition*

Many languages have a small number of reserved words for forming iterative loops. For example, Pascal has `while`, `repeat`, and `for` statements. In contrast, Common Lisp has an almost bewildering range of possibilities, as summarized below:

<code>dolist</code>	loop over elements of a list
<code>dotimes</code>	loop over successive integers
<code>do, do*</code>	general loop, sparse syntax
<code>loop</code>	general loop, verbose syntax
<code>mapc, mapcar</code>	loop over elements of lists(s)
<code>some, every</code>	loop over list until condition
<code>find, reduce, etc.</code>	more specific looping functions
<code>recursion</code>	general repetition

To explain each possibility, we will present versions of the function `length`, which returns the number of elements in a list. First, the special form `dolist` can be used to iterate over the elements of a list. The syntax is:

```
(dolist (variable list optional-result) body...)
```

This means that the body is executed once for each element of the list, with *variable* bound to the first element, then the second element, and so on. At the end, `dolist` evaluates and returns the *optional-result* expression, or `nil` if there is no result expression.

Below is a version of `length` using `dolist`. The `let` form introduces a new variable, `len`, which is initially bound to zero. The `dolist` form then executes the body once for each element of the list, with the body incrementing `len` by one each time. This use is unusual in that the loop iteration variable, `element`, is not used in the body.

```
(defun length1 (list)
  (let ((len 0)) ; start with LEN=0
    (dolist (element list) ; and on each iteration
      (incf len)) ; increment LEN by 1
    len)) ; and return LEN
```

It is also possible to use the optional result of `dolist`, as shown below. While many programmers use this style, I find that it is too easy to lose track of the result, and so I prefer to place the result last explicitly.

```
(defun length1.1 (list) ; alternate version:
  (let ((len 0)) ; (not my preference)
    (dolist (element list len) ; uses len as result here
      (incf len)))
```

The function `mapc` performs much the same operation as the special form `dolist`. In the simplest case, `mapc` takes two arguments, the first a function, the second a list. It applies the function to each element of the list. Here is `length` using `mapc`:

```
(defun length2 (list)
  (let ((len 0)) ; start with LEN=0
    (mapc #'(lambda (element) ; and on each iteration
              (incf len)) ; increment LEN by 1
          list)
    len)) ; and return LEN
```

There are seven different mapping functions, of which the most useful are `mapc` and `mapcar`. `mapcar` executes the same function calls as `mapc`, but then returns the results

in a list.

There is also a `dotimes` form, which has the syntax:

```
(dotimes (variable number optional-result) body...)
```

and executes the body with *variable* bound first to zero, then one, all the way up to *number*-1 (for a total of *number* times). Of course, `dotimes` is not appropriate for implementing `length`, since we don't know the number of iterations ahead of time.

There are two very general looping forms, `do` and `loop`. The syntax of `do` is as follows:

```
(do ((variable initial next)...)
     (exit-test result)
     body...)
```

Each *variable* is initially bound to the *initial* value. If *exit-test* is true, then *result* is returned. Otherwise, the body is executed and each *variable* is set to the corresponding *next* value and *exit-test* is tried again. The loop repeats until *exit-test* is true. If a *next* value is omitted, then the corresponding variable is not updated each time through the loop. Rather, it is treated as if it had been bound with a `let` form.

Here is `length` implemented with `do`, using two variables, `len` to count the number of elements, and `l` to go down the list. This is often referred to as *cdr-ing down a list*, because on each operation we apply the function `cdr` to the list. (Actually, here we have used the more mnemonic name `rest` instead of `cdr`.) Note that the `do` loop has no body! All the computation is done in the variable initialization and stepping, and in the end test.

```
(defun length3 (list)
  (do ((len 0 (+ len 1)) ; start with LEN=0, increment
       (l list (rest l))) ; ... on each iteration
      ((null l) len)) ; (until the end of the list)
```

I find the `do` form a little confusing, because it does not clearly say that we are looping through a list. To see that it is indeed iterating over the list requires looking at both the variable `l` and the end test. Worse, there is no variable that stands for the current element of the list; we would need to say `(first l)` to get at it. Both `dolist` and `mapc` take care of stepping, end testing, and variable naming automatically. They are examples of the “be specific” principle. Because it is so unspecific, `do` will not be used much in this book. However, many good programmers use it, so it is important to know how to read `do` loops, even if you decide never to write one.

The syntax of `loop` is an entire language by itself, and a decidedly non-Lisp-like language it is. Rather than list all the possibilities for `loop`, we will just give examples

here, and refer the reader to *Common Lisp the Language*, 2d edition, or chapter 24.5 for more details. Here are three versions of length using loop:

```
(defun length4 (list)
  (loop for element in list      ; go through each element
        count t))               ;   counting each one

(defun length5 (list)
  (loop for element in list      ; go through each element
        summing 1))              ;   adding 1 each time

(defun length6 (list)
  (loop with len = 0            ; start with LEN=0
        until (null list)       ; and (until end of list)
        for element = (pop list) ; on each iteration
        do (incf len)           ; increment LEN by 1
        finally (return len)))  ; and return LEN
```

Every programmer learns that there are certain kinds of loops that are used again and again. These are often called *programming idioms* or *cliches*. An example is going through the elements of a list or array and doing some operation to each element. In most languages, these idioms do not have an explicit syntactic marker. Instead, they are implemented with a general loop construct, and it is up to the reader of the program to recognize what the programmer is doing.

Lisp is unusual in that it provides ways to explicitly encapsulate such idioms, and refer to them with explicit syntactic and functional forms. dolist and dotimes are two examples of this—they both follow the “be specific” principle. Most programmers prefer to use a dolist rather than an equivalent do, because it cries out “this loop iterates over the elements of a list.” Of course, the corresponding do form also says the same thing—but it takes more work for the reader to discover this.

In addition to special forms like dolist and dotimes, there are quite a few functions that are designed to handle common idioms. Two examples are count-if, which counts the number of elements of a sequence that satisfy a predicate, and position-if, which returns the index of an element satisfying a predicate. Both can be used to implement length. In length7 below, count-if gives the number of elements in list that satisfy the predicate true. Since true is defined to be always true, this gives the length of the list.

```
(defun length7 (list)
  (count-if #'true list))

(defun true (x) t)
```

In length8, the function position-if finds the position of an element that satisfies the predicate true, starting from the end of the list. This will be the very last element

of the list, and since indexing is zero-based, we add one to get the length. Admittedly, this is not the most straightforward implementation of `length`.

```
(defun length8 (list)
  (if (null list)
      0
      (+ 1 (position-if #'true list :from-end t))))
```

A partial table of functions that implement looping idioms is given below. These functions are designed to be flexible enough to handle almost all operations on sequences. The flexibility comes in three forms. First, functions like `mapcar` can apply to an arbitrary number of lists, not just one:

```
> (mapcar #'- '(1 2 3)) => (-1 -2 -3)
> (mapcar #'+ '(1 2) '(10 20)) => (11 22)
> (mapcar #'+ '(1 2) '(10 20) '(100 200)) => (111 222)
```

Second, many of the functions accept keywords that allow the user to vary the test for comparing elements, or to only consider part of the sequence.

```
> (remove 1 '(1 2 3 2 1 0 -1)) => (2 3 2 0 -1)
> (remove 1 '(1 2 3 2 1 0 -1) :key #'abs) => (2 3 2 0)
> (remove 1 '(1 2 3 2 1 0 -1) :test #'<) => (1 1 0 -1)
> (remove 1 '(1 2 3 2 1 0 -1) :start 4) => (1 2 3 2 0 -1)
```

Third, some have corresponding functions ending in `-if` or `-if-not` that take a predicate rather than an element to match against:

```
> (remove-if #'oddp '(1 2 3 2 1 0 -1)) => (2 2 0)
> (remove-if-not #'oddp '(1 2 3 2 1 0 -1)) => (1 3 1 -1)
> (find-if #'evenp '(1 2 3 2 1 0 -1)) => 2
```

The following two tables assume these two values:

```
(setf x '(a b c))
(setf y '(1 2 3))
```

The first table lists functions that work on any number of lists but do not accept keywords:

(every #'oddp y)	$\Rightarrow \text{nil}$	test if every element satisfies a predicate
(some #'oddp y)	$\Rightarrow \text{t}$	test if some element satisfies predicate
(mapcar #'- y)	$\Rightarrow (-1 \ -2 \ -3)$	apply function to each element and return result
(mapc #'print y)	<i>prints 1 2 3</i>	perform operation on each element

The second table lists functions that have `-if` and `-if-not` versions and also accept keyword arguments:

(member 2 y)	$\Rightarrow (2 \ 3)$	see if element is in list
(count 'b x)	$\Rightarrow 1$	count the number of matching elements
(delete 1 y)	$\Rightarrow (2 \ 3)$	omit matching elements
(find 2 y)	$\Rightarrow 2$	find first element that matches
(position 'a x)	$\Rightarrow 0$	find index of element in sequence
(reduce #'+ y)	$\Rightarrow 6$	apply function to successive elements
(remove 2 y)	$\Rightarrow (1 \ 3)$	like delete, but makes a new copy
(substitute 4 2 y)	$\Rightarrow (1 \ 4 \ 3)$	replace elements with new ones

### *Repetition through Recursion*

Lisp has gained a reputation as a “recursive” language, meaning that Lisp encourages programmers to write functions that call themselves. As we have seen above, there is a dizzying number of functions and special forms for writing loops in Common Lisp, but it is also true that many programs handle repetition through recursion rather than with a syntactic loop.

One simple definition of `length` is “the empty list has length 0, and any other list has a length which is one more than the length of the rest of the list (after the first element).” This translates directly into a recursive function:

```
(defun length9 (list)
  (if (null list)
      0
      (+ 1 (length9 (rest list)))))
```

This version of `length` arises naturally from the recursive definition of a list: “a list is either the empty list or an element consed onto another list.” In general, most recursive functions derive from the recursive nature of the data they are operating on. Some kinds of data, like binary trees, are hard to deal with in anything but a recursive fashion. Others, like lists and integers, can be defined either recursively (leading to recursive functions) or as a sequence (leading to iterative functions). In this book, I tend to use the “list-as-sequence” view rather than the “list-as-first-and-rest” view. The reason is that defining a list as a first and a rest is an arbitrary and artificial distinction that is based on the implementation of lists that Lisp happens to use. But there are many other ways to decompose a list. We could break it into the last

element and all-but-the-last elements, for example, or the first half and the second half. The “list-as-sequence” view makes no such artificial distinction. It treats all elements identically.

One objection to the use of recursive functions is that they are inefficient, because the compiler has to allocate memory for each recursive call. This may be true for the function `length9`, but it is not necessarily true for all recursive calls. Consider the following definition:

```
(defun length10 (list)
  (length10-aux list 0))

(defun length10-aux (sublist len-so-far)
  (if (null sublist)
      len-so-far
      (length10-aux (rest sublist) (+ 1 len-so-far))))
```

`length10` uses `length10-aux` as an auxiliary function, passing it 0 as the length of the list so far. `length10-aux` then goes down the list to the end, adding 1 for each element. The invariant relation is that the length of the sublist plus `len-so-far` always equals the length of the original list. Thus, when the sublist is nil, then `len-so-far` is the length of the original list. Variables like `len-so-far` that keep track of partial results are called *accumulators*. Other examples of functions that use accumulators include `flatten-all` on page 329; `one-unknown` on page 237; the Prolog predicates discussed on page 686; and `anonymous-variables-in` on pages 400 and 433, which uses two accumulators.

The important difference between `length9` and `length10` is *when* the addition is done. In `length9`, the function calls itself, then returns, and then adds 1. In `length10-aux`, the function adds 1, then calls itself, then returns. There are no pending operations to do after the recursive call returns, so the compiler is free to release any memory allocated for the original call before making the recursive call. `length10-aux` is called a *tail-recursive* function, because the recursive call appears as the last thing the function does (the tail). Many compilers will optimize tail-recursive calls, although not all do. (Chapter 22 treats tail-recursion in more detail, and points out that Scheme compilers guarantee that tail-recursive calls will be optimized.)

Some find it ugly to introduce `length10-aux`. For them, there are two alternatives. First, we could combine `length10` and `length10-aux` into a single function with an optional parameter:

```
(defun length11 (list &optional (len-so-far 0))
  (if (null list)
      len-so-far
      (length11 (rest list) (+ 1 len-so-far))))
```

Second, we could introduce a *local* function inside the definition of the main function. This is done with the special form `labels`:

```
(defun length12 (the-list)
  (labels
    ((length13 (list len-so-far)
      (if (null list)
          len-so-far
          (length13 (rest list) (+ 1 len-so-far))))))
    (length13 the-list 0)))
```

In general, a `labels` form (or the similar `flet` form) can be used to introduce one or more local functions. It has the following syntax:

```
(labels
  ((function-name (parameter...) function-body)...)
  body-of-labels)
```

### *Other Special Forms*

A few more special forms do not fit neatly into any category. We have already seen the two special forms for creating constants and functions, `quote` and `function`. These are so common that they have abbreviations: '`x`' for `(quote x)` and '#'`f` for `(function f)`.

The special form `progn` can be used to evaluate a sequence of forms and return the value of the last one:

```
(progn (setf x 0) (setf x (+ x 1)) x) ⇒ 1
```

`progn` is the equivalent of a `begin...end` block in other languages, but it is used very infrequently in Lisp. There are two reasons for this. First, programs written in a functional style never need a sequence of actions, because they don't have side effects. Second, even when side effects are used, many special forms allow for a body which is a sequence—an implicit `progn`. I can only think of three places where a `progn` is justified. First, to implement side effects in a branch of a two-branched conditional, one could use either an `if` with a `progn`, or a `cond`:

<pre>(if (&gt; x 100)   (progn (print "too big")          (setf x 100))   x)</pre>	<pre>(cond ((&gt; x 100)        (print "too big")        (setf x 100))       (t x))</pre>
--	---

If the conditional had only one branch, then `when` or `unless` should be used, since they allow an implicit progn. If there are more than two branches, then `cond` should be used.

Second, `progn` is sometimes needed in macros that expand into more than one top-level form, as in the `defun*` macro on page 326, section 10.3. Third, a `progn` is sometimes needed in an `unwind-protect`, an advanced macro. An example of this is the `with-resource` macro on page 338, section 10.4.

The forms `trace` and `untrace` are used to control debugging information about entry and exit to a function:

```
> (trace length9) ⇒ (LENGTH9)
> (length9 '(a b c)) ⇒
(1 ENTER LENGTH9: (A B C))
(2 ENTER LENGTH9: (B C))
(3 ENTER LENGTH9: (C))
(4 ENTER LENGTH9: NIL)
(4 EXIT LENGTH9: 0)
(3 EXIT LENGTH9: 1)
(2 EXIT LENGTH9: 2)
(1 EXIT LENGTH9: 3)
3
> (untrace length9) ⇒ (LENGTH9)
> (length9 '(a b c)) ⇒ 3
```

Finally, the special form `return` can be used to break out of a block of code. Blocks are set up by the special form `block`, or by the looping forms (`do`, `do*`, `dolist`, `dotimes`, or `loop`). For example, the following function computes the product of a list of numbers, but if any number is zero, then the whole product must be zero, so we immediately return zero from the `dolist` loop. Note that this returns from the `dolist` only, not from the function itself (although in this case, the value returned by `dolist` becomes the value returned by the function, because it is the last expression in the function). I have used uppercase letters in `RETURN` to emphasize the fact that it is an unusual step to exit from a loop.

```
(defun product (numbers)
  "Multiply all the numbers together to compute their product."
  (let ((prod 1))
    (dolist (n numbers prod)
      (if (= n 0)
          (RETURN 0)
          (setf prod (* n prod))))))
```

## Macros

The preceding discussion has been somewhat cavalier with the term “special form.” Actually, some of these special forms are really *macros*, forms that the compiler expands into some other code. Common Lisp provides a number of built-in macros and allows the user to extend the language by defining new macros. (There is no way for the user to define new special forms, however.)

Macros are defined with the special form `defmacro`. Suppose we wanted to define a macro, `while`, that would act like the `while` loop statement of Pascal. Writing a macro is a four-step process:

- Decide if the macro is really necessary.
- Write down the syntax of the macro.
- Figure out what the macro should expand into.
- Use `defmacro` to implement the syntax/expansion correspondence.

The first step in writing a macro is to recognize that every time you write one, you are defining a new language that is just like Lisp except for your new macro. The programmer who thinks that way will rightfully be extremely frugal in defining macros. (Besides, when someone asks, “What did you get done today?” it sounds more impressive to say “I defined a new language and wrote a compiler for it” than to say “I just hacked up a couple of macros.”) Introducing a macro puts much more memory strain on the reader of your program than does introducing a function, variable or data type, so it should not be taken lightly. Introduce macros only when there is a clear need, and when the macro fits in well with your existing system. As C.A.R. Hoare put it, “One thing the language designer should not do is to include untried ideas of his own.”

The next step is to decide what code the macro should expand into. It is a good idea to follow established Lisp conventions for macro syntax whenever possible. Look at the looping macros (`dolist`, `dotimes`, `do-symbols`), the defining macros (`defun`, `defvar`, `defparameter`, `defstruct`), or the the I/O macros (`with-open-file`, `with-open-stream`, `with-input-from-string`), for example. If you follow the naming and syntax conventions for one of these instead of inventing your own conventions, you’ll be doing the reader of your program a favor. For `while`, a good syntax is:

```
(while test body...)
```

The third step is to write the code that you want a macro call to expand into:

```
(loop
  (unless test (return nil))
  body)
```

The final step is to write the definition of the macro, using `defmacro`. A `defmacro` form is similar to a `defun` in that it has a parameter list, optional documentation string, and body. There are a few differences in what is allowed in the parameter list, which will be covered later. Here is a definition of the macro `while`, which takes a test and a body, and builds up the `loop` code shown previously:

```
(defmacro while (test &rest body)
  "Repeat body while test is true."
  (list* 'loop
    (list 'unless test '(return nil))
    body))
```

(The function `list*` is like `list`, except that the last argument is appended onto the end of the list of the other arguments.) We can see what this macro expands into by using `macroexpand`, and see how it runs by typing in an example:

```
> (macroexpand-1 '(while (< i 10)
  (print (* i i))
  (setf i (+ i 1)))) =>
(LOOP (UNLESS (< I 10) (RETURN NIL))
  (PRINT (* I I))
  (SETF I (+ I 1)))

> (setf i 7) => 7
> (while (< i 10)
  (print (* i i))
  (setf i (+ i 1))) =>
49
64
81
NIL
```

Section 24.6 (page 853) describes a more complicated macro and some details on the pitfalls of writing complicated macros (page 855).

### *Backquote Notation*

The hardest part about defining `while` is building the code that is the expansion of the macro. It would be nice if there was a more immediate way of building code. The following version of `while` following attempts to do just that. It defines the local

variable code to be a template for the code we want, and then substitutes the real values of the variables test and body for the placeholders in the code. This is done with the function `subst`; (`subst new old tree`) substitutes `new` for each occurrence of `old` anywhere within `tree`.

```
(defmacro while (test &rest body)
  "Repeat body while test is true."
  (let ((code '(loop (unless test (return nil)) . body)))
    (subst test 'test (subst body 'body code))))
```

The need to build up code (and noncode data) from components is so frequent that there is a special notation for it, the *backquote* notation. The backquote character `` `` is similar to the quote character `` '''. A backquote indicates that what follows is *mostly* a literal expression but may contain some components that are to be evaluated. Anything marked by a leading comma `,` is evaluated and inserted into the structure, and anything marked with a leading `,@` must evaluate to a list that is spliced into the structure: each element of the list is inserted, without the top-level parentheses. The notation is covered in more detail in section 23.5. Here we use the combination of backquote and comma to rewrite `while`:

```
(defmacro while (test &rest body)
  "Repeat body while test is true."
  '(loop (unless ,test (return nil))
        ,@body))
```

Here are some more examples of backquote. Note that at the end of a list, `,@` has the same effect as `.` followed by `,`. In the middle of a list, only `,@` is a possibility.

```
> (setf test1 '(a test)) => (A TEST)

> `(this is ,test1) => (THIS IS (A TEST))

> `(this is ,@test1) => (THIS IS A TEST)

> `(this is . ,test1) => (THIS IS A TEST)

> `(this is ,@test1 -- this is only ,@test1) =>
  (THIS IS A TEST -- THIS IS ONLY A TEST)
```

This completes the section on special forms and macros. The remaining sections of this chapter give an overview of the important built-in functions in Common Lisp.

### 3.3 Functions on Lists

For the sake of example, assume we have the following assignments:

```
(setf x '(a b c))
(setf y '(1 2 3))
```

The most important functions on lists are summarized here. The more complicated ones are explained more thoroughly when they are used.

(first x)	$\Rightarrow a$	first element of a list
(second x)	$\Rightarrow b$	second element of a list
(third x)	$\Rightarrow c$	third element of a list
(nth 0 x)	$\Rightarrow a$	nth element of a list, 0-based
(rest x)	$\Rightarrow (b\ c)$	all but the first element
(car x)	$\Rightarrow a$	another name for the first element of a list
(cdr x)	$\Rightarrow (b\ c)$	another name for all but the first element
(last x)	$\Rightarrow (c)$	last cons cell in a list
(length x)	$\Rightarrow 3$	number of elements in a list
(reverse x)	$\Rightarrow (c\ b\ a)$	puts list in reverse order
(cons 0 y)	$\Rightarrow (0\ 1\ 2\ 3)$	add to front of list
(append x y)	$\Rightarrow (a\ b\ c\ 1\ 2\ 3)$	append together elements
(list x y)	$\Rightarrow ((a\ b\ c)\ (1\ 2\ 3))$	make a new list
(list* 1 2 x)	$\Rightarrow (1\ 2\ a\ b\ c)$	append last argument to others
(null nil)	$\Rightarrow T$	predicate is true of the empty list
(null x)	$\Rightarrow \text{nil}$	... and false for everything else
(listp x)	$\Rightarrow T$	predicate is true of any list, including <code>nil</code>
(listp 3)	$\Rightarrow \text{nil}$	... and is false for nonlists
(consp x)	$\Rightarrow t$	predicate is true of non-nil lists
(consp nil)	$\Rightarrow \text{nil}$	... and false for atoms, including <code>nil</code>
(equal x x)	$\Rightarrow t$	true for lists that look the same
(equal x y)	$\Rightarrow \text{nil}$	... and false for lists that look different
(sort y #'>)	$\Rightarrow (3\ 2\ 1)$	sort a list according to a comparison function
(subseq x 1 2)	$\Rightarrow (B)$	subsequence with given start and end points

We said that `(cons a b)` builds a longer list by adding element `a` to the front of list `b`, but what if `b` is not a list? This is not an error; the result is an object `x` such that `(first x)  $\Rightarrow a$` , `(rest x)  $\Rightarrow b$` , and where `x` prints as `(a . b)`. This is known as *dotted pair* notation. If `b` is a list, then the usual list notation is used for output rather than the dotted pair notation. But either notation can be used for input.

So far we have been thinking of lists as sequences, using phrases like “a list of three elements.” The list is a convenient abstraction, but the actual implementation of lists relies on lower-level building blocks called *cons cells*. A cons cell is a data structure with two fields: a first and a rest. What we have been calling “a list of three elements” can also be seen as a single cons cell, whose first field points to

the first element and whose rest field points to another cons cell that is a cons cell representing a list of two elements. This second cons cell has a rest field that is a third cons cell, one whose rest field is nil. All proper lists have a last cons cell whose rest field is nil. Figure 3.1 shows the cons cell notation for the three-element list (one two three), as well as for the result of (cons 'one 'two).

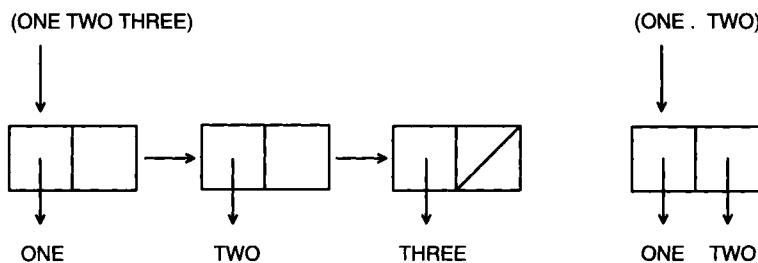


Figure 3.1: Cons Cell Diagrams

- ?** **Exercise 3.2 [s]** The function `cons` can be seen as a special case of one of the other functions listed previously. Which one?
- ?** **Exercise 3.3 [m]** Write a function that will print an expression in dotted pair notation. Use the built-in function `princ` to print each component of the expression.
- ?** **Exercise 3.4 [m]** Write a function that, like the regular `print` function, will print an expression in dotted pair notation when necessary but will use normal list notation when possible.

## 3.4 Equality and Internal Representation

In Lisp there are five major equality predicates, because not all objects are created equally equal. The numeric equality predicate, `=`, tests if two numbers are the same. It is an error to apply `=` to non-numbers. The other equality predicates operate on any kind of object, but to understand the difference between them, we need to understand some of the internals of Lisp.

When Lisp reads a symbol in two different places, the result is guaranteed to be the exact same symbol. The Lisp system maintains a symbol table that the function `read` uses to map between characters and symbols. But when a list is read (or built)

in two different places, the results are *not* identically the same, even though the corresponding elements may be. This is because read calls cons to build up the list, and each call to cons returns a new cons cell. Figure 3.2 shows two lists, x and y, which are both equal to (one two), but which are composed of different cons cells, and hence are not identical. Figure 3.3 shows that the expression (rest x) does not generate new cons cells, but rather shares structure with x, and that the expression (cons 'zero x) generates exactly one new cons cell, whose rest is x.

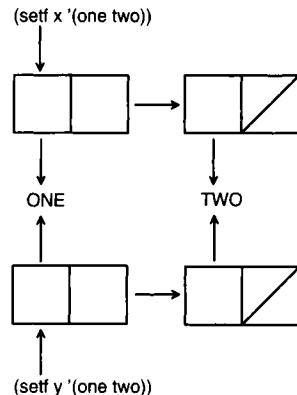


Figure 3.2: Equal But Nonidentical Lists

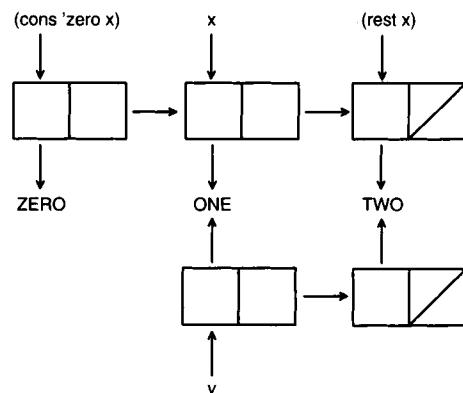


Figure 3.3: Parts of Lists

When two mathematically equal numbers are read (or computed) in two places, they may or may not be the same, depending on what the designers of your implementation felt was more efficient. In most systems, two equal fixnums will be identical, but equal numbers of other types will not (except possibly short floats). Common Lisp provides four equality predicates of increasing generality. All four begin with the letters `eq`, with more letters meaning the predicate considers more objects to be equal. The simplest predicate is `eq`, which tests for the exact same object. Next, `eql` tests for objects that are either `eq` or are equivalent numbers. `equal` tests for objects that are either `eql` or are lists or strings with `eql` elements. Finally, `equalp` is like `equal` except it also matches uppercase and lowercase characters and numbers of different types. The following table summarizes the results of applying each of the four predicates to various values of `x` and `y`. The ? value means that the result depends on your implementation: two integers that are `eql` may or may not be `eq`.

<code>x</code>	<code>y</code>	<code>eq</code>	<code>eql</code>	<code>equal</code>	<code>equalp</code>
'x	'x	T	T	T	T
'0	'0	?	T	T	T
'(x)	'(x)	nil	nil	T	T
'"xy"	'"xy"	nil	nil	T	T
'"Xy"	'"xY"	nil	nil	nil	T
'0	'0.0	nil	nil	nil	T
'0	'1	nil	nil	nil	nil

In addition, there are specialized equality predicates such as `=`, `tree-equal`, `char-equal`, and `string-equal`, which compare numbers, trees, characters, and strings, respectively.

## 3.5 Functions on Sequences

Common Lisp is in a transitional position halfway between the Lisps of the past and the Lisps of the future. Nowhere is that more apparent than in the sequence functions. The earliest Lisps dealt only with symbols, numbers, and lists, and provided list functions like `append` and `length`. More modern Lisps added support for vectors, strings, and other data types, and introduced the term *sequence* to refer to both vectors and lists. (A vector is a one-dimensional array: It can be represented more compactly than a list, because there is no need to store the rest pointers. It is also more efficient to get at the *n*th element of a vector, because there is no need to follow a chain of pointers.) Modern Lisps also support strings that are vectors of characters, and hence also a subtype of sequence.

With the new data types came the problem of naming functions that operated on them. In some cases, Common Lisp chose to extend an old function: `length` can

apply to vectors as well as lists. In other cases, the old names were reserved for the list functions, and new names were invented for generic sequence functions. For example, `append` and `mapcar` only work on lists, but `concatenate` and `map` work on any kind of sequence. In still other cases, new functions were invented for specific data types. For example, there are seven functions to pick the  $n$ th element out of a sequence. The most general is `elt`, which works on any kind of sequence, but there are specific functions for lists, arrays, strings, bit vectors, simple bit vectors, and simple vectors. Confusingly, `nth` is the only one that takes the index as the first argument:

```
(nth n list)
(elt sequence n)
(aref array n)
(char string n)
(bit bit vector n)
(sbit simple-bit vector n)
(svref simple-vector n)
```

The most important sequence functions are listed elsewhere in this chapter, depending on their particular purpose.

## 3.6 Functions for Maintaining Tables

Lisp lists can be used to represent a one-dimensional sequence of objects. Because they are so versatile, they have been put to other purposes, such as representing tables of information. The *association list* is a type of list used to implement tables. An association list is a list of dotted pairs, where each pair consists of a *key* and a *value*. Together, the list of pairs form a table: given a key, we can retrieve the corresponding value from the table, or verify that there is no such key stored in the table. Here's an example for looking up the names of states by their two-letter abbreviation. The function `assoc` is used. It returns the key/value pair (if there is one). To get the value, we just take the `cdr` of the result returned by `assoc`.

```
(setf state-table
  '((AL . Alabama) (AK . Alaska) (AZ . Arizona) (AR . Arkansas)))
> (assoc 'AK state-table) => (AK . ALASKA)
> (cdr (assoc 'AK state-table)) => ALASKA
> (assoc 'TX state-table) => NIL
```

If we want to search the table by value rather than by key, we can use `rassoc`:

```
> (rassoc 'Arizona table) => (AZ . ARIZONA)
```

```
> (car (rassoc 'Arizona table)) ⇒ AZ
```

Managing a table with `assoc` is simple, but there is one drawback: we have to search through the whole list one element at a time. If the list is very long, this may take a while.

Another way to manage tables is with *hash tables*. These are designed to handle large amounts of data efficiently but have a degree of overhead that can make them inappropriate for small tables. The function `gethash` works much like `get`—it takes two arguments, a key and a table. The table itself is initialized with a call to `make-hash-table` and modified with a `setf` of `gethash`:

```
(setf table (make-hash-table))

(setf (gethash 'AL table) 'Alabama)
(setf (gethash 'AK table) 'Alaska)
(setf (gethash 'AZ table) 'Arizona)
(setf (gethash 'AR table) 'Arkansas)
```

Here we retrieve values from the table:

```
> (gethash 'AK table) ⇒ ALASKA
> (gethash 'TX table) ⇒ NIL
```

The function `remhash` removes a key/value pair from a hash table, `clrhash` removes all pairs, and `maphash` can be used to map over the key/value pairs. The keys to hash tables are not restricted; they can be any Lisp object. There are many more details on the implementation of hash tables in Common Lisp, and an extensive literature on their theory.

A third way to represent table is with *property lists*. A property list is a list of alternating key/value pairs. Property lists (sometimes called p-lists or plists) and association lists (sometimes called a-lists or alists) are similar:

```
a-list: ((key1 . val1) (key2 . val2) ... (keyn . valn))
p-list: (key1 val1 key2 val2 ... keyn valn)
```

Given this representation, there is little to choose between a-lists and p-lists. They are slightly different permutations of the same information. The difference is in how they are normally used. Every symbol has a property list associated with it. That means we can associate a property/value pair directly with a symbol. Most programs use only a few different properties but have many instances of property/value pairs for each property. Thus, each symbol's p-list will likely be short. In our example, we are only interested in one property: the state associated with each abbreviation.

That means that the property lists will be very short indeed: one property for each abbreviation, instead of a list of 50 pairs in the association list implementation.

Property values are retrieved with the function `get`, which takes two arguments: the first is a symbol for which we are seeking information, and the second is the property of that symbol that we are interested in. `get` returns the value of that property, if one has been stored. Property/value pairs can be stored under a symbol with a `setf` form. A table would be built as follows:

```
(setf (get 'AL 'state) 'Alabama)
(setf (get 'AK 'state) 'Alaska)
(setf (get 'AZ 'state) 'Arizona)
(setf (get 'AR 'state) 'Arkansas)
```

Now we can retrieve values with `get`:

```
> (get 'AK 'state) => ALASKA
> (get 'TX 'state) => NIL
```

This will be faster because we can go immediately from a symbol to its lone property value, regardless of the number of symbols that have properties. However, if a given symbol has more than one property, then we still have to search linearly through the property list. As Abraham Lincoln might have said, you can make some of the table lookups faster some of the time, but you can't make all the table lookups faster all of the time. Notice that there is no equivalent of `rassoc` using property lists; if you want to get from a state to its abbreviation, you could store the abbreviation under a property of the state, but that would be a separate `setf` form, as in:

```
(setf (get 'Arizona 'abbrev) 'AZ)
```

In fact, when source, property, and value are all symbols, there are quite a few possibilities for how to use properties. We could have mimicked the a-list approach, and listed all the properties under a single symbol, using `setf` on the function `symbol-plist` (which gives a symbol's complete property list):

```
(setf (symbol-plist 'state-table)
      '(AL Alabama AK Alaska AZ Arizona AR Arkansas))
> (get 'state-table 'AL) => ALASKA
> (get 'state-table 'Alaska) => NIL
```

Property lists have a long history in Lisp, but they are falling out of favor as new alternatives such as hash tables are introduced. There are two main reasons why property lists are avoided. First, because symbols and their property lists are global,

it is easy to get conflicts when trying to put together two programs that use property lists. If two programs use the same property for different purposes, they cannot be used together. Even if two programs use *different* properties on the same symbols, they will slow each other down. Second, property lists are messy. There is no way to remove quickly every element of a table implemented with property lists. In contrast, this can be done trivially with clrhash on hash tables, or by setting an association list to nil.

## 3.7 Functions on Trees

Many Common Lisp functions treat the expression ((a b) ((c)) (d e)) as a sequence of three elements, but there are a few functions that treat it as a tree with five non-null leaves. The function copy-tree creates a copy of a tree, and tree-equal tests if two trees are equal by traversing cons cells, but not other complex data like vectors or strings. In that respect, tree-equal is similar to equal, but tree-equal is more powerful because it allows a :test keyword:

```
> (setf tree '((a b) ((c)) (d e)))
> (tree-equal tree (copy-tree tree)) => T
(defun same-shape-tree (a b)
  "Are two trees the same except for the leaves?"
  (tree-equal a b :test #'true))
(defun true (&rest ignore) t)
> (same-shape-tree tree '((1 2) ((3)) (4 5))) => T
> (same-shape-tree tree '((1 2) (3) (4 5))) => NIL
```

Figure 3.4 shows the tree ((a b) ((c)) (d e)) as a cons cell diagram.

There are also two functions for substituting a new expression for an old one anywhere within a tree. subst substitutes a single value for another, while sublis takes a list of substitutions in the form of an association list of (old . new) pairs. Note that the order of old and new in the a-list for sublis is reversed from the order of arguments to subst. The name sublis is uncharacteristically short and confusing; a better name would be subst-list.

```
> (subst 'new 'old '(old ((very old)))) => (NEW ((VERY NEW)))
> (sublis '((old . new)) '(old ((very old)))) => (NEW ((VERY NEW)))
> (subst 'new 'old 'old) => 'NEW
```

```
(defun english->french (words)
  (sublis '((are . va) (book . libre) (friend . ami)
            (hello . bonjour) (how . comment) (my . mon)
            (red . rouge) (you . tu))
          words))

> (english->french '(hello my friend - how are you today?)) =>
(BONJOUR MON AMI - COMMENT VA TU TODAY?)
```

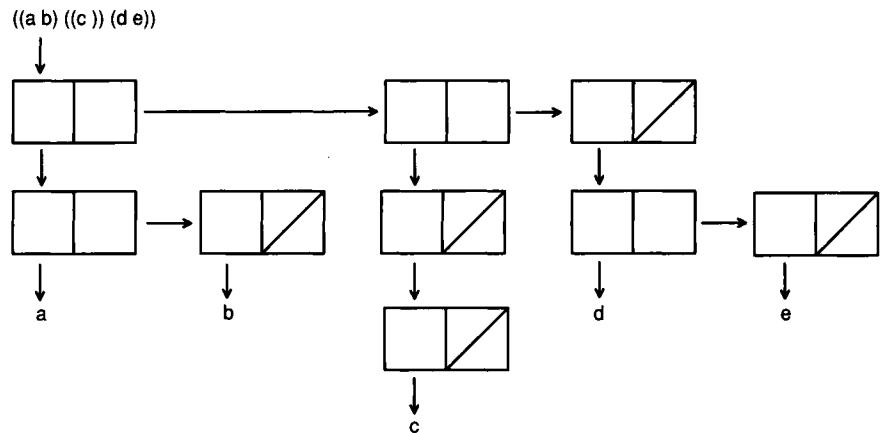


Figure 3.4: Cons Cell Diagram of a Tree

## 3.8 Functions on Numbers

The most commonly used functions on numbers are listed here. There are quite a few other numeric functions that have been omitted.

(+ 4 2)	$\Rightarrow 6$	add
(- 4 2)	$\Rightarrow 2$	subtract
(* 4 2)	$\Rightarrow 8$	multiply
(/ 4 2)	$\Rightarrow 2$	divide
(> 100 99)	$\Rightarrow t$	greater than (also $\geq$ , greater than or equal to)
(= 100 100)	$\Rightarrow t$	equal (also $\neq$ , not equal)
(< 99 100)	$\Rightarrow t$	less than (also $\leq$ , less than or equal to)
(random 100)	$\Rightarrow 42$	random integer from 0 to 99
(expt 4 2)	$\Rightarrow 16$	exponentiation (also $\exp$ , $e^x$ and $\log$ )
(sin pi)	$\Rightarrow 0.0$	sine function (also $\cos$ , $\tan$ , etc.)
(asin 0)	$\Rightarrow 0.0$	arcsine or $\sin^{-1}$ function (also $\acos$ , $\atan$ , etc.)
(min 2 3 4)	$\Rightarrow 2$	minimum (also $\max$ )
(abs -3)	$\Rightarrow 3$	absolute value
(sqrt 4)	$\Rightarrow 2$	square root
(round 4.1)	$\Rightarrow 4$	round off (also $\text{truncate}$ , $\text{floor}$ , $\text{ceiling}$ )
(rem 11 5)	$\Rightarrow 1$	remainder (also $\text{mod}$ )

## 3.9 Functions on Sets

One of the important uses of lists is to represent sets. Common Lisp provides functions that treat lists in just that way. For example, to see what elements the sets  $r = \{a, b, c, d\}$  and  $s = \{c, d, e\}$  have in common, we could use:

```
> (setf r '(a b c d))  $\Rightarrow$  (A B C D)
> (setf s '(c d e))  $\Rightarrow$  (C D E)
> (intersection r s)  $\Rightarrow$  (C D)
```

This implementation returned (C D) as the answer, but another might return (D C). They are equivalent sets, so either is valid, and your program should not depend on the order of elements in the result. Here are the main functions on sets:

(intersection r s)	$\Rightarrow$ (c d)	find common elements of two sets
(union r s)	$\Rightarrow$ (a b c d e)	find all elements in either of two sets
(set-difference r s)	$\Rightarrow$ (a b)	find elements in one but not other set
(member 'd r)	$\Rightarrow$ (d)	check if an element is a member of a set
(subsetp s r)	$\Rightarrow$ nil	see if all elements of one set are in another
(adjoin 'b s)	$\Rightarrow$ (b c d e)	add an element to a set
(adjoin 'c s)	$\Rightarrow$ (c d e)	... but don't add duplicates

It is also possible to represent a set with a sequence of bits, given a particular universe of discourse. For example, if every set we are interested in must be a subset of (a b c d e), then we can use the bit sequence 11110 to represent (a b c d), 00000 to represent the empty set, and 11001 to represent (a b e). The bit sequence can be represented in Common Lisp as a bit vector, or as an integer in binary notation. For example, (a b e) would be the bit vector #\*11001 or the integer 25, which can also be written as #b11001.

The advantage of using bit sequences is that it takes less space to encode a set, assuming a small universe. Computation will be faster, because the computer's underlying instruction set will typically process 32 elements at a time.

Common Lisp provides a full complement of functions on both bit vectors and integers. The following table lists some, their correspondence to the list functions.

lists	integers	bit vectors
intersection	logand	bit-and
union	logior	bit-ior
set-difference	logandc2	bit-andc2
member	logbitp	bit
length	logcount	

For example,

```
(intersection '(a b c d) '(a b e)) ⇒ (A B)
(bit-and      #*11110      #*11001)  ⇒ #*11000
(logand      #b11110      #b11001)  ⇒ 24 ≡ #b11000
```

## 3.10 Destructive Functions

In mathematics, a function is something that computes an output value given some input arguments. Functions do not "do" anything, they just compute results. For example, if I tell you that  $x = 4$  and  $y = 5$  and ask you to apply the function "plus" to  $x$  and  $y$ , I expect you to tell me 9. If I then ask, "Now what is the value of  $x$ ?" it would be surprising if  $x$  had changed. In mathematics, applying an operator to  $x$  can have no effect on the value of  $x$ .

In Lisp, some functions *are* able to take effect beyond just computing the result. These "functions" are not functions in the mathematical sense,<sup>2</sup> and in other languages they are known as "procedures." Of course, most of the Lisp functions *are* true mathematical functions, but the few that are not can cause great problems. They can

---

<sup>2</sup>In mathematics, a function must associate a unique output value with each input value.

also be quite useful in certain situations. For both reasons, they are worth knowing about.

Consider the following:

```
> (setf x '(a b c)) => (A B C)
> (setf y '(1 2 3)) => (1 2 3)
> (append x y) => (A B C 1 2 3)
```

append is a pure function, so after evaluating the call to append, we can rightfully expect that x and y retain their values. Now consider this:

```
> (nconc x y) => (A B C 1 2 3)
> x => (A B C 1 2 3)
> y => (1 2 3)
```

The function nconc computes the same result as append, but it has the side effect of altering its first argument. It is called a *destructive* function, because it destroys existing structures, replacing them with new ones. This means that there is quite a conceptual load on the programmer who dares to use nconc. He or she must be aware that the first argument may be altered, and plan accordingly. This is far more complicated than the case with nondestructive functions, where the programmer need worry only about the results of a function call.

The advantage of nconc is that it doesn't use any storage. While append must make a complete copy of x and then have that copy end with y, nconc does not need to copy anything. Instead, it just changes the rest field of the last element of x to point to y. So use destructive functions when you need to conserve storage, but be aware of the consequences.

Besides nconc, many of the destructive functions have names that start with n, including nreverse, nintersection, nunion, nset-difference, and nsubst. An important exception is delete, which is the name used for the destructive version of remove. Of course, the setf special form can also be used to alter structures, but it is the destructive functions that are most dangerous, because it is easier to overlook their effects.



**Exercise 3.5 [h]** (Exercise in altering structure.) Write a program that will play the role of the guesser in the game Twenty Questions. The user of the program will have in mind any type of thing. The program will ask questions of the user, which must be answered yes or no, or "it" when the program has guessed it. If the program runs out of guesses, it gives up and asks the user what "it" was. At first the program will not play well, but each time it plays, it will remember the user's replies and use them for subsequent guesses.

## 3.11 Overview of Data Types

This chapter has been organized around functions, with similar functions grouped together. But there is another way of organizing the Common Lisp world: by considering the different data types. This is useful for two reasons. First, it gives an alternative way of seeing the variety of available functionality. Second, the data types themselves are objects in the Common Lisp language, and as we shall see, there are functions that manipulate data types. These are useful mainly for testing objects (as with the `typecase` macro) and for making declarations.

Here is a table of the most commonly used data types:

Type	Example	Explanation
character	<code>#\c</code>	A single letter, number, or punctuation mark.
number	42	The most common numbers are floats and integers.
float	3.14159	A number with a decimal point.
integer	42	A whole number, of either fixed or indefinite size:
fixnum	123	An integer that fits in a single word of storage.
bignum	123456789	An integer of unbounded size.
function	<code>'sin</code>	A function can be applied to an argument list.
symbol	<code>sin</code>	Symbols can name fns and vars, and are themselves objects.
null	<code>nil</code>	The object <code>nil</code> is the only object of type null.
keyword	<code>:key</code>	Keywords are a subtype of symbol.
sequence	<code>(a b c)</code>	Sequences include lists and vectors.
list	<code>(a b c)</code>	A list is either a cons or null.
vector	<code>#{(a b c)}</code>	A vector is a subtype of sequence.
cons	<code>(a b c)</code>	A cons is a non-nil list.
atom	<code>t</code>	An atom is anything that is not a cons.
string	<code>"abc"</code>	A string is a type of vector of characters.
array	<code>#1A(a b c)</code>	Arrays include vectors and higher-dimensional arrays.
structure	<code>#S(type ...)</code>	Structures are defined by defstruct.
hash-table	<code>...</code>	Hash tables are created by <code>make-hash-table</code> .

Almost every data type has a *recognizer predicate*—a function that returns true for only elements of that type. In general, a predicate is a function that always returns one of two values: true or false. In Lisp, the false value is `nil`, and every other value is considered true, although the most common true value is `t`. In most cases, the recognizer predicate's name is composed of the type name followed by `p`: `characterp` recognizes characters, `numberp` recognizes numbers, and so on. For example, `(numberp 3)` returns `t` because 3 is a number, but `(numberp "x")` returns `nil` because "x" is a string, not a number.

Unfortunately, Common Lisp is not completely regular. There are no recognizers for fixnums, bignums, sequences, and structures. Two recognizers, `null` and `atom`, do not end in `p`. Also note that there is a hyphen before the `p` in `hash-table-p`, because the type has a hyphen in it. In addition, all the recognizers generated by `defstruct` have a hyphen before the `p`.

The function `type-of` returns the type of its argument, and `typep` tests if an object is of a specified type. The function `subtypep` tests if one type can be determined to be a subtype of another. For example:

```
> (type-of 123) ⇒ FIXNUM
> (typep 123 'fixnum) ⇒ T
> (typep 123 'number) ⇒ T
> (typep 123 'integer) ⇒ T
> (typep 123.0 'integer) ⇒ NIL
> (subtypep 'fixnum 'number) ⇒ T
```

The hierarchy of types is rather complicated in Common Lisp. As the prior example shows, there are many different numeric types, and a number like 123 is considered to be of type `fixnum`, `integer`, and `number`. We will see later that it is also of type `rational` and `t`.

The type hierarchy forms a graph, not just a tree. For example, a vector is both a sequence and an array, although neither array nor sequence are subtypes of each other. Similarly, `null` is a subtype of both `symbol` and `list`.

The following table shows a number of more specialized data types that are not used as often:

Type	Example	Explanation
<code>t</code>	42	Every object is of type <code>t</code> .
<code>null</code>		No object is of type <code>null</code> .
<code>complex</code>	<code>#C(0 1)</code>	Imaginary numbers.
<code>bit</code>	0	Zero or one.
<code>rational</code>	<code>2/3</code>	Rationals include integers and ratios.
<code>ratio</code>	<code>2/3</code>	Exact fractional numbers.
<code>simple-array</code>	<code>#1A(x y)</code>	An array that is not displaced or adjustable.
<code>readtable</code>	<code>...</code>	A mapping from characters to their meanings to read.
<code>package</code>	<code>...</code>	A collection of symbols that form a module.
<code>pathname</code>	<code>#P"/usr/spool/mail"</code>	A file or directory name.
<code>stream</code>	<code>...</code>	A pointer to an open file; used for reading or printing.
<code>random-state</code>	<code>...</code>	A state used as a seed by <code>random</code> .

In addition, there are even more specialized types, such as `short-float`, `compiled-function`, and `bit-vector`. It is also possible to construct more exact types, such as `(vector (integer 0 3) 100)`, which represents a vector of 100 elements, each of which is an integer from 0 to 3, inclusive. Section 10.1 gives more information on types and their use.

While almost every type has a predicate, it is also true that there are predicates that are not type recognizers but rather recognize some more general condition. For

example, `oddp` is true only of odd integers, and `string-greaterp` is true if one string is alphabetically greater than another.

## 3.12 Input/Output

Input in Lisp is incredibly easy because a complete lexical and syntactic parser is available to the user. The parser is called `read`. It is used to read and return a single Lisp expression. If you can design your application so that it reads Lisp expressions, then your input worries are over. Note that the expression parsed by `read` need not be a legal *evaluable* Lisp expression. That is, you can read (`"hello"` `cons` `zzz`) just as well as (`(+ 2 2)`). In cases where Lisp expressions are not adequate, the function `read-char` reads a single character, and `read-line` reads everything up to the next newline and returns it as a string.

To read from the terminal, the functions `read`, `read-char`, or `read-line` (with no arguments) return an expression, a character, and a string up to the end of line, respectively. It is also possible to read from a file. The function `open` or the macro `with-open-stream` can be used to open a file and associate it with a *stream*, Lisp's name for a descriptor of an input/output source. All three `read` functions take three optional arguments. The first is the stream to read from. The second, if true, causes an error to be signaled at end of file. If the second argument is nil, then the third argument indicates the value to return at end of file.

Output in Lisp is similar to output in other languages, such as C. There are a few low-level functions to do specific kinds of output, and there is a very general function to do formatted output. The function `print` prints any object on a new line, with a space following it. `prin1` will print any object without the new line and space. For both functions, the object is printed in a form that could be processed by `read`. For example, the string `"hello there"` would print as `"hello there"`. The function `princ` is used to print in a human-readable format. The string in question would print as `hello there` with `princ`—the quote marks are not printed. This means that `read` cannot recover the original form; `read` would interpret it as two symbols, not one string. The function `write` accepts eleven different keyword arguments that control whether it acts like `prin1` or `princ`, among other things.

The output functions also take a stream as an optional argument. In the following, we create the file `"test.text"` and print two expressions to it. Then we open the file for reading, and try to read back the first expression, a single character, and then two more expressions. Note that the `read-char` returns the character `#\G`, so the following `read` reads the characters `OODBYE` and turns them into a symbol. The final `read` hits the end of file, and so returns the specified value, `eof`.

```
> (with-open-file (stream "test.text" :direction :output)
  (print '(hello there) stream)
  (princ 'goodbye stream)) =>
GOODBYE      ; and creates the file test.text

> (with-open-file (stream "test.text" :direction :input)
  (list (read stream) (read-char stream) (read stream)
    (read stream nil 'eof))) =>
((HELLO THERE) #\G OODBYE EOF)
```

The function `terpri` stands for “terminate print line,” and it skips to the next line. The function `fresh-line` also skips to the next line, unless it can be determined that the output is already at the start of a line.

Common Lisp also provides a very general function for doing formatted output, called `format`. The first argument to `format` is always the stream to print to; use `t` to print to the terminal. The second argument is the format string. It is printed out verbatim, except for *format directives*, which begin with the character “`~`”. These directives tell how to print out the remaining arguments. Users of C’s `printf` function or FORTRAN’s `format` statement should be familiar with this idea. Here’s an example:

```
> (format t "hello, world")
hello, world
NIL
```

Things get interesting when we put in additional arguments and include format directives:

```
> (format t "~&~a plus ~s is ~f" "two" "two" 4)
two plus "two" is 4.0
NIL
```

The directive “`~&`” moves to a fresh line, “`~a`” prints the next argument as `princ` would, “`~s`” prints the next argument as `print` would, and “`~f`” prints a number in floating-point format. If the argument is not a number, then `princ` is used. `format` always returns nil. There are 26 different format directives. Here’s a more complex example:

```
> (let ((numbers '(1 2 3 4 5)))
  (format t "~&{~r~^ plus ~} is ~@r"
          numbers (apply #'+ numbers)))
one plus two plus three plus four plus five is XV
NIL
```

The directive “`~r`” prints the next argument, which should be a number, in English,

and "`~@r`" prints a number as a roman numeral. The compound directive "`~{ ... ~}`" takes the next argument, which must be a list, and formats each element of the list according to the format string inside the braces. Finally, the directive "`~~`" exits from the enclosing "`~{ ... ~}`" loop if there are no more arguments. You can see that format, like `loop`, comprises almost an entire programming language, which, also like `loop`, is not a very Lisplike language.

## 3.13 Debugging Tools

In many languages, there are two strategies for debugging: (1) edit the program to insert print statements, recompile, and try again, or (2) use a debugging program to investigate (and perhaps alter) the internal state of the running program.

Common Lisp admits both these strategies, but it also offers a third: (3) add annotations that are not part of the program but have the effect of automatically altering the running program. The advantage of the third strategy is that once you are done you don't have to go back and undo the changes you would have introduced in the first strategy. In addition, Common Lisp provides functions that display information about the program. You need not rely solely on looking at the source code.

We have already seen how `trace` and `untrace` can be used to provide debugging information (page 65). Another useful tool is `step`, which can be used to halt execution before each subform is evaluated. The form (`(step expression)`) will evaluate and return `expression`, but pauses at certain points to allow the user to inspect the computation, and possibly change things before proceeding to the next step. The commands available to the user are implementation-dependent, but typing a `?` should give you a list of commands. As an example, here we step through an expression twice, the first time giving commands to stop at each subevaluation, and the second time giving commands to skip to the next function call. In this implementation, the commands are control characters, so they do not show up in the output. All output, including the symbols `⇐` and `⇒` are printed by the stepper itself; I have added no annotation.

```
> (step (+ 3 4 (* 5 6 (/ 7 8))))
⇐ (+ 3 4 (* 5 6 (/ 7 8)))
⇐ 3 ⇒ 3
⇐ 4 ⇒ 4
⇐ (* 5 6 (/ 7 8))
⇐ 5 ⇒ 5
⇐ 6 ⇒ 6
⇐ (/ 7 8)
⇐ 7 ⇒ 7
⇐ 8 ⇒ 8
⇐ (/ 7 8) ⇒ 7/8
```

```

 $\Leftarrow (* 5 6 (/ 7 8)) \Rightarrow 105/4$ 
 $\Leftarrow (+ 3 4 (* 5 6 (/ 7 8))) \Rightarrow 133/4$ 
133/4

> (step (+ 3 4 (* 5 6 (/ 7 8))))
 $\Leftarrow (+ 3 4 (* 5 6 (/ 7 8)))$ 
/: 7 8 \Rightarrow 7/8
*: 5 6 7/8 \Rightarrow 105/4
+: 3 4 105/4 \Rightarrow 133/4
 $\Leftarrow (+ 3 4 (* 5 6 (/ 7 8))) \Rightarrow 133/4$ 
133/4

```

The functions `describe`, `inspect`, `documentation`, and `apropos` provide information about the state of the current program. `apropos` prints information about all symbols whose name matches the argument:

```

> (apropos 'string)
MAKE-STRING           function (LENGTH &KEY INITIAL-ELEMENT)
PRIN1-TO-STRING       function (OBJECT)
PRINC-TO-STRING       function (OBJECT)
STRING                function (X)
.
.
```

Once you know what object you are interested in, `describe` can give more information on it:

```

> (describe 'make-string)
Symbol MAKE-STRING is in LISP package.
The function definition is #<FUNCTION MAKE-STRING -42524322>:
  NAME:      MAKE-STRING
  ARGLIST:   (LENGTH &KEY INITIAL-ELEMENT)
  DOCUMENTATION: "Creates and returns a string of LENGTH elements,
all set to INITIAL-ELEMENT."
  DEFINITION: (LAMBDA (LENGTH &KEY INITIAL-ELEMENT)
                      (MAKE-ARRAY LENGTH :ELEMENT-TYPE 'CHARACTER
                                    :INITIAL-ELEMENT (OR INITIAL-ELEMENT
                                    #\SPACE)))
MAKE-STRING has property INLINE: INLINE
MAKE-STRING has property :SOURCE-FILE: #P"SYS:KERNEL; STRINGS"

> (describe 1234.56)
1234.56 is a single-precision floating-point number.
  Sign 0, exponent #o211, 23-bit fraction #o6450754

```

If all you want is a symbol's documentation string, the function `documentation` will do the trick:

```
> (documentation 'first 'function) => "Return the first element of LIST."  
> (documentation 'pi 'variable) => "pi"
```

If you want to look at and possibly alter components of a complex structure, then `inspect` is the tool. In some implementations it invokes a fancy, window-based browser.

Common Lisp also provides a debugger that is entered automatically when an error is signalled, either by an inadvertant error or by deliberate action on the part of the program. The details of the debugger vary between implementations, but there are standard ways of entering it. The function `break` enters the debugger after printing an optional message. It is intended as the primary method for setting debugging break points. `break` is intended only for debugging purposes; when a program is deemed to be working, all calls to `break` should be removed. However, it is still a good idea to check for unusual conditions with `error`, `cerror`, `assert`, or `check-type`, which will be described in the following section.

## 3.14 Antibugging Tools

It is a good idea to include *antibugging* checks in your code, in addition to doing normal debugging. Antibugging code checks for errors and possibly takes corrective action.

The functions `error` and `cerror` are used to signal an error condition. These are intended to remain in the program even after it has been debugged. The function `error` takes a format string and optional arguments. It signals a fatal error; that is, it stops the program and does not offer the user any way of restarting it. For example:

```
(defun average (numbers)  
  (if (null numbers)  
      (error "Average of the empty list is undefined.")  
      (/ (reduce #'+ numbers)  
          (length numbers))))
```

In many cases, a fatal error is a little drastic. The function `cerror` stands for continuable error. `cerror` takes two format strings; the first prints a message indicating what happens if we continue, and the second prints the error message itself. `cerror` does not actually take any action to repair the error, it just allows the user to signal that continuing is alright. In the following implementation, the user continues by typing `:continue`. In ANSI Common Lisp, there are additional ways of specifying options for continuing.

```
(defun average (numbers)
  (if (null numbers)
      (progn
        (cerror "Use 0 as the average."
               "Average of the empty list is undefined.")
        0)
      (/ (reduce #'+ numbers)
          (length numbers)))))

> (average '())
Error: Average of the empty list is undefined.
Error signaled by function AVERAGE.
If continued: Use 0 as the average.
>> :continue
0
```

In this example, adding error checking nearly doubled the length of the code. This is not unusual; there is a big difference between code that works on the expected input and code that covers all possible errors. Common Lisp tries to make it easier to do error checking by providing a few special forms. The form `ecase` stands for “exhaustive case” or “error case.” It is like a normal case form, except that if none of the cases are satisfied, an error message is generated. The form `ccase` stands for “continuable case.” It is like `ecase`, except that the error is continuable. The system will ask for a new value for the test object until the user supplies one that matches one of the programmed cases.

To make it easier to include error checks without inflating the length of the code too much, Common Lisp provides the special forms `check-type` and `assert`. As the name implies, `check-type` is used to check the type of an argument. It signals a continuable error if the argument has the wrong type. For example:

```
(defun sqr (x)
  "Multiply x by itself."
  (check-type x number)
  (* x x))
```

If `sqr` is called with a non-number argument, an appropriate error message is printed:

```
> (sqr "hello")
Error: the argument X was "hello", which is not a NUMBER.
If continued: replace X with new value
>> :continue 4
16
```

`assert` is more general than `check-type`. In the simplest form, `assert` tests an

expression and signals an error if it is false. For example:

```
(defun sqr (x)
  "Multiply x by itself."
  (assert (numberp x))
  (* x x))
```

There is no possibility of continuing from this kind of assertion. It is also possible to give `assert` a list of places that can be modified in an attempt to make the assertion true. In this example, the variable `x` is the only thing that can be changed:

```
(defun sqr (x)
  "Multiply x by itself."
  (assert (numberp x) (x))
  (* x x))
```

If the assertion is violated, an error message will be printed and the user will be given the option of continuing by altering `x`. If `x` is given a value that satisfies the assertion, then the program continues. `assert` always returns nil.

Finally, the user who wants more control over the error message can provide a format control string and optional arguments. So the most complex syntax for `assert` is:

```
(assert test-form (place...) format-ctl-string format-arg...)
```

Here is another example. The assertion tests that the temperature of the bear's porridge is neither too hot nor too cold.

```
(defun eat-porridge (bear)
  (assert (< too-cold (temperature (bear-porridge bear)) too-hot)
         (bear (bear-porridge bear)))
         "~a's porridge is not just right: ~a"
         bear (hotness (bear-porridge bear)))
  (eat (bear-porridge bear)))
```

In the interaction below, the assertion failed, and the programmer's error message was printed, along with two possibilities for continuing. The user selected one, typed in a call to `make-porridge` for the new value, and the function successfully continued.

```
> (eat-porridge momma-bear)
Error: #<MOMMA BEAR>'s porridge is not just right: 39
Restart actions (select using :continue):
 0: Supply a new value for BEAR
 1: Supply a new value for (BEAR-PORRIDGE BEAR)
>> :continue 1
Form to evaluate and use to replace (BEAR-PORRIDGE BEAR):
(make-porridge :temperature just-right)
nil
```

It may seem like wasted effort to spend time writing assertions that (if all goes well) will never be used. However, for all but the perfect programmer, bugs do occur, and the time spent antibugging will more than pay for itself in saving debugging time.

Whenever you develop a complex data structure, such as some kind of data base, it is a good idea to develop a corresponding consistency checker. A consistency checker is a function that will look over a data structure and test for all possible errors. When a new error is discovered, a check for it should be incorporated into the consistency checker. Calling the consistency checker is the fastest way to help isolate bugs in the data structure.

In addition, it is a good idea to keep a list of difficult test cases on hand. That way, when the program is changed, it will be easy to see if the change reintroduces a bug that had been previously removed. This is called *regression testing*, and Waters (1991) presents an interesting tool for maintaining a suite of regression tests. But it is simple enough to maintain an informal test suite with a function that calls assert on a series of examples:

```
(defun test-ex ()
  "Test the program EX on a series of examples."
  (init-ex) ; Initialize the EX program first.
  (assert (equal (ex 3 4)    5))
  (assert (equal (ex 5 0)    0))
  (assert (equal (ex 'x 0)   0)))
```

### *Timing Tools*

A program is not complete just because it gives the right output. It must also deliver the output in a timely fashion. The form (*time expression*) can be used to see how long it takes to execute *expression*. Some implementations also print statistics on the amount of storage required. For example:

```
> (defun f (n) (dotimes (i n) nil)) => F
```

```
> (time (f 10000)) => NIL
Evaluation of (F 10000) took 4.347272 Seconds of elapsed time,
including 0.0 seconds of paging time for 0 faults, Consed 27 words.

> (compile 'f) => F

> (time (f 10000)) => NIL
Evaluation of (F 10000) took 0.011518 Seconds of elapsed time,
including 0.0 seconds of paging time for 0 faults, Consed 0 words.
```

This shows that the compiled version is over 300 times faster and uses less storage to boot. Most serious Common Lisp programmers work exclusively with compiled functions. However, it is usually a bad idea to worry too much about efficiency details while starting to develop a program. It is better to design a flexible program, get it to work, and then modify the most frequently used parts to be more efficient. In other words, separate the development stage from the fine-tuning stage. Chapters 9 and 10 give more details on efficiency consideration, and chapter 25 gives more advice on debugging and antibugging techniques.

## 3.15 Evaluation

There are three functions for doing evaluation in Lisp: `funcall`, `apply`, and `eval`. `funcall` is used to apply a function to individual arguments, while `apply` is used to apply a function to a list of arguments. Actually, `apply` can be given one or more individual arguments before the final argument, which is always a list. `eval` is passed a single argument, which should be an entire form—a function or special form followed by its arguments, or perhaps an atom. The following five forms are equivalent:

```
> (+ 1 2 3 4)           => 10
> (funcall #'+ 1 2 3 4) => 10
> (apply #'+ '(1 2 3 4)) => 10
> (apply #'+ 1 2 '(3 4)) => 10
> (eval '(+ 1 2 3 4))   => 10
```

In the past, `eval` was seen as the key to Lisp's flexibility. In modern Lisps with lexical scoping, such as Common Lisp, `eval` is used less often (in fact, in Scheme there is no `eval` at all). Instead, programmers are expected to use `lambda` to create a new function, and then `apply` or `funcall` the function. In general, if you find yourself using `eval`, you are probably doing the wrong thing.

## 3.16 Closures

What does it mean to create a new function? Certainly every time a function (or '#') special form is evaluated, a function is returned. But in the examples we have seen and in the following one, it is always the *same* function that is returned.

```
> (mapcar #'(lambda (x) (+ x x)) '(1 3 10)) ⇒ (2 6 20)
```

Every time we evaluate the #'(lambda ...) form, it returns the function that doubles its argument. However, in the general case, a function consists of the body of the function coupled with any *free lexical variables* that the function references. Such a pairing is called a *lexical closure*, or just a *closure*, because the lexical variables are enclosed within the function. Consider this example:

```
(defun adder (c)
  "Return a function that adds c to its argument."
  #'(lambda (x) (+ x c)))

> (mapcar (adder 3) '(1 3 10)) ⇒ (4 6 13)
> (mapcar (adder 10) '(1 3 10)) ⇒ (11 13 20)
```

Each time we call adder with a different value for c, it creates a different function, the function that adds c to its argument. Since each call to adder creates a new local variable named c, each function returned by adder is a unique function.

Here is another example. The function bank-account returns a closure that can be used as a representation of a bank account. The closure captures the local variable balance. The body of the closure provides code to access and modify the local variable.

```
(defun bank-account (balance)
  "Open a bank account starting with the given balance."
  #'(lambda (action amount)
      (case action
        (deposit (setf balance (+ balance amount)))
        (withdraw (setf balance (- balance amount))))))
```

In the following, two calls to bank-account create two different closures, each with a separate value for the lexical variable balance. The subsequent calls to the two closures change their respective balances, but there is no confusion between the two accounts.

```
> (setf my-account (bank-account 500.00)) ⇒ #<CLOSURE 52330407>
```

```
> (setf your-account (bank-account 250.00)) => #<CLOSURE 52331203>
> (funcall my-account 'withdraw 75.00) => 425.0
> (funcall your-account 'deposit 250.00) => 500.0
> (funcall your-account 'withdraw 100.00) => 400.0
> (funcall my-account 'withdraw 25.00) => 400.0
```

This style of programming will be considered in more detail in chapter 13.

## 3.17 Special Variables

Common Lisp provides for two kinds of variables: *lexical* and *special* variables. For the beginner, it is tempting to equate the special variables in Common Lisp with global variables in other languages. Unfortunately, this is not quite correct and can lead to problems. It is best to understand Common Lisp variables on their own terms.

By default, Common Lisp variables are *lexical variables*. Lexical variables are introduced by some syntactic construct like `let` or `defun` and get their name from the fact that they may only be referred to by code that appears lexically within the body of the syntactic construct. The body is called the *scope* of the variable.

So far, there is no difference between Common Lisp and other languages. The interesting part is when we consider the *extent*, or lifetime, of a variable. In other languages, the extent is the same as the scope: a new local variable is created when a block is entered, and the variable goes away when the block is exited. But because it is possible to create new functions—closures—in Lisp, it is therefore possible for code that references a variable to live on after the scope of the variable has been exited. Consider again the `bank-account` function, which creates a closure representing a bank account:

```
(defun bank-account (balance)
  "Open a bank account starting with the given balance."
  #'(lambda (action amount)
      (case action
        (deposit (setf balance (+ balance amount)))
        (withdraw (setf balance (- balance amount))))))
```

The function introduces the lexical variable `balance`. The scope of `balance` is the body of the function, and therefore references to `balance` can occur only within this scope. What happens when `bank-account` is called and exited? Once the body of the function has been left, no other code can refer to that instance of `balance`. The scope has been exited, but the extent of `balance` lives on. We can call the closure, and it

can reference `balance`, because the code that created the closure appeared lexically within the scope of `balance`.

In summary, Common Lisp lexical variables are different because they can be captured inside closures and referred to even after the flow of control has left their scope.

Now we will consider special variables. A variable is made special by a `defvar` or `defparameter` form. For example, if we say

```
(defvar *counter* 0)
```

then we can refer to the special variable `*counter*` anywhere in our program. This is just like a familiar global variable. The tricky part is that the global binding of `*counter*` can be shadowed by a local binding for that variable. In most languages, the local binding would introduce a local lexical variable, but in Common Lisp, special variables can be bound both locally and globally. Here is an example:

```
(defun report ()
  (format t "Counter = ~d " *counter*))

> (report)
Counter = 0
NIL

> (let ((*counter* 100))
  (report))
Counter = 100
NIL

> (report)
Counter = 0
NIL
```

There are three calls to `report` here. In the first and third, `report` prints the global value of the special variable `*counter*`. In the second call, the `let` form introduces a new binding for the special variable `*counter*`, which is again printed by `report`. Once the scope of the `let` is exited, the new binding is disestablished, so the final call to `report` uses the global value again.

In summary, Common Lisp special variables are different because they have global scope but admit the possibility of local (dynamic) shadowing. Remember: A lexical variable has lexical scope and indefinite extent. A special variable has indefinite scope and dynamic extent.

The function call (`(symbol-value var)`), where `var` evaluates to a symbol, can be used to get at the current value of a special variable. To set a special variable, the following two forms are completely equivalent:

```
(setf (symbol-value var) value)
      (set var value)
```

where both *var* and *value* are evaluated. There are no corresponding forms for accessing and setting lexical variables. Special variables set up a mapping between symbols and values that is accessible to the running program. This is unlike lexical variables (and all variables in traditional languages) where symbols (identifiers) have significance only while the program is being compiled. Once the program is running, the identifiers have been compiled away and cannot be used to access the variables; only code that appears within the scope of a lexical variable can reference that variable.

 **Exercise 3.6 [s]** Given the following initialization for the lexical variable *a* and the special variable *\*b\**, what will be the value of the *let* form?

```
(setf a 'global-a)
      (defvar *b* 'global-b)

      (defun fn () *b*)

      (let ((a 'local-a)
            (*b* 'local-b))
        (list a *b* (fn) (symbol-value 'a) (symbol-value '*b*)))
```

## 3.18 Multiple Values

Throughout this book we have spoken of “the value returned by a function.” Historically, Lisp was designed so that every function returns a value, even those functions that are more like procedures than like functions. But sometimes we want a single function to return more than one piece of information. Of course, we can do that by making up a list or structure to hold the information, but then we have to go to the trouble of defining the structure, building an instance each time, and then taking that instance apart to look at the pieces. Consider the function *round*. One way it can be used is to round off a floating-point number to the nearest integer. So (*round* 5.1) is 5. Sometimes, though not always, the programmer is also interested in the fractional part. The function *round* serves both interested and disinterested programmers by returning two values: the rounded integer and the remaining fraction:

```
> (round 5.1) => 5 .1
```

There are two values after the  $\Rightarrow$  because *round* returns two values. Most of the time,

multiple values are ignored, and only the first value is used. So `(* 2 (round 5.1))` is 10, just as if `round` had only returned a single value. If you want to get at multiple values, you have to use a special form, such as `multiple-value-bind`:

```
(defun show-both (x)
  (multiple-value-bind (int rem)
    (round x)
    (format t "~f = ~d + ~f" x int rem)))

> (show-both 5.1)
5.1 = 5 + 0.1
```

You can write functions of your own that return multiple values using the function `values`, which returns its arguments as multiple values:

```
> (values 1 2 3) => 1 2 3
```

Multiple values are a good solution because they are unobtrusive until they are needed. Most of the time when we are using `round`, we are only interested in the integer value. If `round` did not use multiple values, if it packaged the two values up into a list or structure, then it would be harder to use in the normal cases.

It is also possible to return no values from a function with `(values)`. This is sometimes used by procedures that are called for effect, such as printing. For example, `describe` is defined to print information and then return no values:

```
> (describe 'x)
Symbol X is in the USER package.
It has no value, definition or properties.
```

However, when `(values)` or any other expression returning no values is nested in a context where a value is expected, it still obeys the Lisp rule of one-value-per-expression and returns `nil`. In the following example, `describe` returns no values, but then `list` in effect asks for the first value and gets `nil`.

```
> (list (describe 'x))
Symbol X is in AILP package.
It has no value, definition or properties.
(NIL)
```

## 3.19 More about Parameters

Common Lisp provides the user with a lot of flexibility in specifying the parameters to a function, and hence the arguments that the function accepts. Following is a program that gives practice in arithmetic. It asks the user a series of  $n$  problems, where each problem tests the arithmetic operator  $op$  (which can be  $+$ ,  $-$ ,  $*$ , or  $/$ , or perhaps another binary operator). The arguments to the operator will be random integers from 0 to  $range$ . Here is the program:

```
(defun math-quiz (op range n)
  "Ask the user a series of math problems."
  (dotimes (i n)
    (problem (random range) op (random range)))

  (defun problem (x op y)
    "Ask a math problem, read a reply, and say if it is correct."
    (format t "~&How much is ~d ~a ~d?" x op y)
    (if (eql (read) (funcall op x y))
        (princ "Correct!")
        (princ "Sorry, that's not right.")))
```

and here is an example of its use:

```
> (math-quiz '+ 100 2)
How much is 32 + 60? 92
Correct!
How much is 91 + 19? 100
Sorry, that's not right.
```

One problem with the function `math-quiz` is that it requires the user to type three arguments: the operator, a range, and the number of iterations. The user must remember the order of the arguments, and remember to quote the operator. This is quite a lot to expect from a user who presumably is just learning to add!

Common Lisp provides two ways of dealing with this problem. First, a programmer can specify that certain arguments are *optional*, and provide default values for those arguments. For example, in `math-quiz` we can arrange to make `+` be the default operator, 100 be the default number range, and 10 be the default number of examples with the following definition:

```
(defun math-quiz (&optional (op '+) (range 100) (n 10))
  "Ask the user a series of math problems."
  (dotimes (i n)
    (problem (random range) op (random range))))
```

Now (`math-quiz`) means the same as (`math-quiz + 100 10`). If an optional parameter appears alone without a default value, then the default is `nil`. Optional parameters are handy; however, what if the user is happy with the operator and range but wants to change the number of iterations? Optional parameters are still position-dependent, so the only solution is to type in all three arguments: (`math-quiz + 100 5`).

Common Lisp also allows for parameters that are position-independent. These *keyword* parameters are explicitly named in the function call. They are useful when there are a number of parameters that normally take default values but occasionally need specific values. For example, we could have defined `math-quiz` as:

```
(defun math-quiz (&key (op '+) (range 100) (n 10))
  "Ask the user a series of math problems."
  (dotimes (i n)
    (problem (random range) op (random range))))
```

Now (`math-quiz :n 5`) and (`math-quiz :op '+' :n 5 :range 100`) mean the same. Keyword arguments are specified by the parameter name preceded by a colon, and followed by the value. The keyword/value pairs can come in any order.

A symbol starting with a colon is called a *keyword*, and can be used anywhere, not just in argument lists. The term *keyword* is used differently in Lisp than in many other languages. For example, in Pascal, keywords (or *reserved words*) are syntactic symbols, like `if`, `else`, `begin`, and `end`. In Lisp we call such symbols *special form operators* or just *special forms*. Lisp keywords are symbols that happen to reside in the keyword package.<sup>3</sup> They have no special syntactic meaning, although they do have the unusual property of being self-evaluating: they are constants that evaluate to themselves, unlike other symbols, which evaluate to whatever value was stored in the variable named by the symbol. Keywords also happen to be used in specifying `&key` argument lists, but that is by virtue of their value, not by virtue of some syntax rule. It is important to remember that keywords are used in the function call, but normal nonkeyword symbols are used as parameters in the function definition.

Just to make things a little more confusing, the symbols `&optional`, `&rest`, and `&key` are called *lambda-list keywords*, for historical reasons. Unlike the colon in real keywords, the `&` in lambda-list keywords has no special significance. Consider these annotated examples:

---

<sup>3</sup>A *package* is a symbol table: a mapping between strings and the symbols they name.

```

> :xyz => :XYZ                                ; keywords are self-evaluating
> &optional =>                               ; lambda-list keywords are normal symbols
Error: the symbol &optional has no value

> '&optional => &OPTIONAL

> (defun f (&xyz) (+ &xyz &xyz)) => F ; & has no significance

> (f 3) => 6

> (defun f (:xyz) (+ :xyz :xyz)) =>
Error: the keyword :xyz appears in a variable list.
Keywords are constants, and so cannot be used as names of variables.

> (defun g (&key x y) (list x y)) => G

> (let ((keys '(:x :y :z)))                  ; keyword args can be computed
  (g (second keys) 1 (first keys) 2)) => (2 1)

```

Many of the functions presented in this chapter take keyword arguments that make them more versatile. For example, remember the function `find`, which can be used to look for a particular element in a sequence:

```
> (find 3 '(1 2 3 4 -5 6.0)) => 3
```

It turns out that `find` takes several optional keyword arguments. For example, suppose we tried to find 6 in this sequence:

```
> (find 6 '(1 2 3 4 -5 6.0)) => nil
```

This fails because `find` tests for equality with `eq`, and 6 is not `eq` to 6.0. However, 6 is `equalp` to 6.0, so we could use the `:test` keyword:

```
> (find 6 '(1 2 3 4 -5 6.0) :test #'equalp) => 6.0
```

In fact, we can specify any binary predicate for the `:test` keyword; it doesn't have to be an equality predicate. For example, we could find the first number that 4 is less than:

```
> (find 4 '(1 2 3 4 -5 6.0) :test #'<) => 6.0
```

Now suppose we don't care about the sign of the numbers; if we look for 5, we want to find the -5. We can handle this with the `key` keyword to take the absolute value of each element of the list with the `abs` function:

```
> (find 5 '(1 2 3 4 -5 6.0) :key #'abs) => -5
```

Keyword parameters significantly extend the usefulness of built-in functions, and they can do the same for functions you define. Among the built-in functions, the most common keywords fall into two main groups: :test, :test-not and :key, which are used for matching functions, and :start, :end, and :from-end, which are used on sequence functions. Some functions accept both sets of keywords. (*Common Lisp the Language*, 2d edition, discourages the use of :test-not keywords, although they are still a part of the language.)

The matching functions include `sublis`, `position`, `subst`, `union`, `intersection`, `set-difference`, `remove`, `remove-if`, `subsetp`, `assoc`, `find`, and `member`. By default, each tests if some item is `eql` to one or more of a series of other objects. This test can be changed by supplying some other predicate as the argument to :test, or it can be reversed by specifying :test-not. In addition, the comparison can be made against some part of the object rather than the whole object by specifying a selector function as the :key argument.

The sequence functions include `remove`, `remove-if`, `position`, and `find`. The most common type of sequence is the list, but strings and vectors can also be used as sequences. A sequence function performs some action repeatedly for some elements of a sequence. The default is to go through the sequence from beginning to end, but the reverse order can be specified with :from-end t, and a subsequence can be specified by supplying a number for the :start or :end keyword. The first element of a sequence is numbered 0, not 1, so be careful.

As an example of keyword parameters, suppose we wanted to write sequence functions that are similar to `find` and `find-if`, except that they return a list of all matching elements rather than just the first matching element. We will call the new functions `find-all` and `find-all-if`. Another way to look at these functions is as variations of `remove`. Instead of removing items that match, they keep all the items that match, and remove the ones that don't. Viewed this way, we can see that the function `find-all-if` is actually the same function as `remove-if-not`. It is sometimes useful to have two names for the same function viewed in different ways (like `not` and `null`). The new name could be defined with a `defun`, but it is easier to just copy over the definition:

```
(setf (symbol-function 'find-all-if) #'remove-if-not)
```

Unfortunately, there is no built-in function that corresponds exactly to `find-all`, so we will have to define it. Fortunately, `remove` can do most of the work. All we have to do is arrange to pass `remove` the complement of the :test predicate. For example, finding all elements that are equal to 1 in a list is equivalent to removing elements that are not equal to 1:

```
> (setf nums '(1 2 3 2 1)) ⇒ (1 2 3 2 1)

> (find-all 1 nums :test #'=) ≡ (remove 1 nums :test #'/=) ⇒ (1 1)
```

Now what we need is a higher-order function that returns the complement of a function. In other words, given `=`, we want to return `/=`. This function is called `complement` in ANSI Common Lisp, but it was not defined in earlier versions, so it is given here:

```
(defun complement (fn)
  "If FN returns y, then (complement FN) returns (not y)."
  ;; This function is built-in in ANSI Common Lisp.
  ;; but is defined here for those with non-ANSI compilers.
  #'(lambda (&rest args) (not (apply fn args))))
```

When `find-all` is called with a given `:test` predicate, all we have to do is call `remove` with the complement as the `:test` predicate. This is true even when the `:test` function is not specified, and therefore defaults to `eql`. We should also test for when the user specifies the `:test-not` predicate, which is used to specify that the match succeeds when the predicate is false. It is an error to specify both a `:test` and `:test-not` argument to the same call, so we need not test for that case. The definition is:

```
(defun find-all (item sequence &rest keyword-args
                    &key (test #'eql) test-not &allow-other-keys)
  "Find all those elements of sequence that match item,
  according to the keywords. Doesn't alter sequence."
  (if test-not
      (apply #'remove item sequence
             :test-not (complement test-not) keyword-args)
      (apply #'remove item sequence
             :test (complement test) keyword-args)))
```

The only hard part about this definition is understanding the parameter list. The `&rest` accumulates all the keyword/value pairs in the variable `keyword-args`. In addition to the `&rest` parameter, two specific keyword parameters, `:test` and `:test-not`, are specified. Any time you put a `&key` in a parameter list, you need an `&allow-other-keys` if, in fact, other keywords are allowed. In this case we want to accept keywords like `:start` and `:key` and pass them on to `remove`.

All the keyword/value pairs will be accumulated in the list `keyword-args`, including the `:test` or `:test-not` values. So we will have:

```
(find-all 1 nums :test #'= :key #'abs)
≡ (remove 1 nums :test (complement #'=) :test #'= :key #'abs)
⇒ (1 1)
```

Note that the call to `remove` will contain two `:test` keywords. This is not an error; Common Lisp declares that the leftmost value is the one that counts.

 **Exercise 3.7 [s]** Why do you think the leftmost of two keys is the one that counts, rather than the rightmost?

 **Exercise 3.8 [m]** Some versions of Kyoto Common Lisp (KCL) have a bug wherein they use the rightmost value when more than one keyword/value pair is specified for the same keyword. Change the definition of `find-all` so that it works in KCL.

There are two more lambda-list keywords that are sometimes used by advanced programmers. First, within a macro definition (but not a function definition), the symbol `&body` can be used as a synonym for `&rest`. The difference is that `&body` instructs certain formatting programs to indent the rest as a body. Thus, if we defined the macro:

```
(defmacro while2 (test &body body)
  "Repeat body while test is true."
  '(loop (if (not ,test) (return nil))
        . ,body))
```

Then the automatic indentation of `while2` (on certain systems) is prettier than `while`:

<pre>(while (&lt; i 10)       (print (* i i))       (setf i (+ i 1)))</pre>	<pre>(while2 (&lt; i 10)         (print (* i i))         (setf i (+ i 1)))</pre>
---	--

Finally, an `&aux` can be used to bind a new local variable or variables, as if bound with `let*`. Personally, I consider this an abomination, because `&aux` variables are not parameters at all and thus have no place in a parameter list. I think they should be clearly distinguished as local variables with a `let`. But some good programmers do use `&aux`, presumably to save space on the page or screen. Against my better judgement, I show an example:

```
(defun length14 (list &aux (len 0))
  (dolist (element list len)
    (incf len)))
```

## 3.20 The Rest of Lisp

There is a lot more to Common Lisp than what we have seen here, but this overview should be enough for the reader to comprehend the programs in the chapters to come. The serious Lisp programmer will further his or her education by continuing to consult reference books and online documentation. You may also find part V of this book to be helpful, particularly chapter 24, which covers advanced features of Common Lisp (such as packages and error handling) and chapter 25, which is a collection of troubleshooting hints for the perplexed Lisper.

While it may be distracting for the beginner to be continually looking at some reference source, the alternative—to explain every new function in complete detail as it is introduced—would be even more distracting. It would interrupt the description of the AI programs, which is what this book is all about.

## 3.21 Exercises

-  **Exercise 3.9 [m]** Write a version of `length` using the function `reduce`.
-  **Exercise 3.10 [m]** Use a reference manual or describe to figure out what the functions `lcm` and `nreconc` do.
-  **Exercise 3.11 [m]** There is a built-in Common Lisp function that, given a key, a value, and an association list, returns a new association list that is extended to include the key/value pair. What is the name of this function?
-  **Exercise 3.12 [m]** Write a single expression using `format` that will take a list of words and print them as a sentence, with the first word capitalized and a period after the last word. You will have to consult a reference to learn new `format` directives.

## 3.22 Answers

**Answer 3.2**  $(\text{cons } a \ b) \equiv (\text{list}^* \ a \ b)$

**Answer 3.3**

```
(defun dprint (x)
  "Print an expression in dotted pair notation."
  (cond ((atom x) (princ x))
        (t (princ "(")
            (dprint (first x))
            (pr-rest (rest x)))
            (princ ")"))
        x)))

(defun pr-rest (x)
  (princ " . ")
  (dprint x))
```

**Answer 3.4** Use the same `dprint` function defined in the last exercise, but change `pr-rest`.

```
(defun pr-rest (x)
  (cond ((null x))
        ((atom x) (princ " . ") (princ x))
        (t (princ " ") (dprint (first x)) (pr-rest (rest x))))))
```

**Answer 3.5** We will keep a data base called `*db*`. The data base is organized into a tree structure of nodes. Each node has three fields: the name of the object it represents, a node to go to if the answer is yes, and a node for when the answer is no. We traverse the nodes until we either get an “it” reply or have to give up. In the latter case, we destructively modify the data base to contain the new information.

```
(defstruct node
  name
  (yes nil)
  (no nil))

(defvar *db*
  (make-node :name 'animal
    :yes (make-node :name 'mammal)
    :no (make-node
      :name 'vegetable
      :no (make-node :name 'mineral))))
```

```
(defun questions (&optional (node *db*))
  (format t "Is it a ~a? " (node-name node))
  (case (read)
    ((y yes) (if (not (null (node-yes node)))
                  (questions (node-yes node))
                  (setf (node-yes node) (give-up))))
     ((n no) (if (not (null (node-no node)))
                  (questions (node-no node))
                  (setf (node-no node) (give-up))))
      (it 'aha!)
      (t (format t "Reply with YES, NO, or IT if I have guessed it.")
          (questions node))))
  (defun give-up ()
    (format t "I give up - what is it? ")
    (make-node :name (read))))
```

Here it is used:

```
> (questions)
Is it a ANIMAL? yes
Is it a MAMMAL? yes
I give up - what is it? bear
#S(NODE :NAME BEAR)

> (questions)
Is it a ANIMAL? yes
Is it a MAMMAL? no
I give up - what is it? penguin
#S(NODE :NAME PENGUIN)

> (questions)
Is it a ANIMAL? yes
Is it a MAMMAL? yes
Is it a BEAR? it
AHA!
```

**Answer 3.6** The value is (LOCAL-A LOCAL-B LOCAL-B GLOBAL-A LOCAL-B).

The let form binds `a` lexically and `*b*` dynamically, so the references to `a` and `*b*` (including the reference to `*b*` within `fn`) all get the local values. The function `symbol-value` always treats its argument as a special variable, so it ignores the lexical binding for `a` and returns the global binding instead. However, the `symbol-value` of `*b*` is the local dynamic value.

**Answer 3.7** There are two good reasons: First, it makes it faster to search through the argument list: just search until you find the key, not all the way to the end. Second, in the case where you want to override an existing keyword and pass the argument list on to another function, it is cheaper to cons the new keyword/value pair on the front of a list than to append it to the end of a list.

**Answer 3.9**

```
(defun length-r (list)
  (reduce #'+ (mapcar #'(lambda (x) 1) list)))
```

or more efficiently:

```
(defun length-r (list)
  (reduce #'(lambda (x y) (+ x 1)) list
         :initial-value 0))
```

or, with an ANSI-compliant Common Lisp, you can specify a :key

```
(defun length-r (list)
  (reduce #'+ list :key #'(lambda (x) 1)))
```

**Answer 3.12** (format t "~@(~{~a~~^ ~}.~)" '(this is a test))

## CHAPTER 4

---

# GPS: The General Problem Solver

*There are now in the world machines that think.*

—Herbert Simon  
Nobel Prize-winning AI researcher

The General Problem Solver, developed in 1957 by Alan Newell and Herbert Simon, embodied a grandiose vision: a single computer program that could solve *any* problem, given a suitable description of the problem. GPS caused quite a stir when it was introduced, and some people in AI felt it would sweep in a grand new era of intelligent machines. Simon went so far as to make this statement about his creation:

*It is not my aim to surprise or shock you. . . . But the simplest way I can summarize is to say that there are now in the world machines that think, that learn and create. Moreover, their ability to do these things is going to increase rapidly until—in a visible future—the range of problems they can handle will be coextensive with the range to which the human mind has been applied.*

Although GPS never lived up to these exaggerated claims, it was still an important program for historical reasons. It was the first program to separate its problem-solving strategy from its knowledge of particular problems, and it spurred much further research in problem solving. For all these reasons, it is a fitting object of study.

The original GPS program had a number of minor features that made it quite complex. In addition, it was written in an obsolete low-level language, IPL, that added gratuitous complexity. In fact, the confusing nature of IPL was probably an important reason for the grand claims about GPS. If the program was that complicated, it *must* do something important. We will be ignoring some of the subtleties of the original program, and we will use Common Lisp, a much more perspicuous language than IPL. The result will be a version of GPS that is quite simple, yet illustrates some important points about AI.

On one level, this chapter is about GPS. But on another level, it is about the process of developing an AI computer program. We distinguish five stages in the development of a program. First is the problem description, which is a rough idea—usually written in English prose—of what we want to do. Second is the program specification, where we redscribe the problem in terms that are closer to a computable procedure. The third stage is the implementation of the program in a programming language such as Common Lisp, the fourth is testing, and the fifth is debugging and analysis. The boundaries between these stages are fluid, and the stages need not be completed in the order stated. Problems at any stage can lead to a change in the previous stage, or even to complete redesign or abandonment of the project. A programmer may prefer to complete only a partial description or specification, proceed directly to implementation and testing, and then return to complete the specification based on a better understanding.

We follow all five stages in the development of our versions of GPS, with the hope that the reader will understand GPS better and will also come to understand better how to write a program of his or her own. To summarize, the five stages of an AI programming project are:

1. **Describe** the problem in vague terms
2. **Specify** the problem in algorithmic terms
3. **Implement** the problem in a programming language
4. **Test** the program on representative examples
5. **Debug and analyze** the resulting program, and repeat the process

## 4.1 Stage 1: Description

As our problem description, we will start with a quote from Newell and Simon's 1972 book, *Human Problem Solving*:

*The main methods of GPS jointly embody the heuristic of means-ends analysis. Means-ends analysis is typified by the following kind of common-sense argument:*

*I want to take my son to nursery school. What's the difference between what I have and what I want? One of distance. What changes distance? My automobile. My automobile won't work. What is needed to make it work? A new battery. What has new batteries? An auto repair shop. I want the repair shop to put in a new battery; but the shop doesn't know I need one. What is the difficulty? One of communication. What allows communication? A telephone . . . and so on.*

*The kind of analysis—classifying things in terms of the functions they serve and oscillating among ends, functions required, and means that perform them—forms the basic system of heuristic of GPS.*

Of course, this kind of analysis is not exactly new. The theory of means-ends analysis was laid down quite elegantly by Aristotle 2300 years earlier in the chapter entitled "The nature of deliberation and its objects" of the *Nicomachean Ethics* (Book III. 3, 1112b):

*We deliberate not about ends, but about means. For a doctor does not deliberate whether he shall heal, nor an orator whether he shall persuade, nor a statesman whether he shall produce law and order, nor does any one else deliberate about his end. They assume the end and consider how and by what means it is attained; and if it seems to be produced by several means they consider by which it is most easily and best produced, while if it is achieved by one only they consider how it will be achieved by this and by what means this will be achieved, till they come to the first cause, which in the order of discovery is last . . . and what is last in the order of analysis seems to be first in the order of becoming. And if we come on an impossibility, we give up the search, e.g., if we need money and this cannot be got; but if a thing appears possible we try to do it.*

Given this description of a theory of problem solving, how should we go about writing a program? First, we try to understand more fully the procedure outlined in the quotes. The main idea is to solve a problem using a process called means-ends analysis, where the problem is stated in terms of what we want to happen. In Newell and Simon's example, the problem is to get the kid to school, but in general we would

like the program to be able to solve a broad class of problems. We can solve a problem if we can find some way to eliminate “the difference between what I have and what I want.” For example, if what I have is a child at home, and what I want is a child at school, then driving may be a solution, because we know that driving leads to a change in location. We should be aware that using means-ends analysis is a choice: it is also possible to start from the current situation and search forward to the goal, or to employ a mixture of different search strategies.

Some actions require the solving of *preconditions* as subproblems. Before we can drive the car, we need to solve the subproblem of getting the car in working condition. It may be that the car is already working, in which case we need do nothing to solve the subproblem. So a problem is solved either by taking appropriate action directly, or by first solving for the preconditions of an appropriate action and then taking the action. It is clear we will need some description of allowable actions, along with their preconditions and effects. We will also need to develop a definition of appropriateness. However, if we can define these notions better, it seems we won’t need any new notions. Thus, we will arbitrarily decide that the problem description is complete, and move on to the problem specification.

## 4.2 Stage 2: Specification

At this point we have an idea—admittedly vague—of what it means to solve a problem in GPS. We can refine these notions into representations that are closer to Lisp as follows:

- We can represent the current state of the world—“what I have”—or the goal state—“what I want”—as sets of conditions. Common Lisp doesn’t have a data type for sets, but it does have lists, which can be used to implement sets. Each condition can be represented by a symbol. Thus, a typical goal might be the list of two conditions (`rich famous`), and a typical current state might be (`unknown poor`).
- We need a list of allowable operators. This list will be constant over the course of a problem, or even a series of problems, but we want to be able to change it and tackle a new problem domain.
- An operator can be represented as a structure composed of an action, a list of preconditions, and a list of effects. We can place limits on the kinds of possible effects by saying that an effect either adds or deletes a condition from the current state. Thus, the list of effects can be split into an add-list and a delete-list. This was the approach taken by the STRIPS<sup>1</sup> implementation of

---

<sup>1</sup>STRIPS is the Stanford Research Institute Problem Solver, designed by Richard Fikes and Nils Nilsson (1971).

GPS, which we will be in effect reconstructing in this chapter. The original GPS allowed more flexibility in the specification of effects, but flexibility leads to inefficiency.

- A complete problem is described to GPS in terms of a starting state, a goal state, and a set of known operators. Thus, GPS will be a function of three arguments. For example, a sample call might be:

```
(GPS '(unknown poor) '(rich famous) list-of-ops)
```

In other words, starting from the state of being poor and unknown, achieve the state of being rich and famous, using any combination of the known operators. GPS should return a true value only if it solves the problem, and it should print a record of the actions taken. The simplest approach is to go through the conditions in the goal state one at a time and try to achieve each one. If they can all be achieved, then the problem is solved.

- A single goal condition can be achieved in two ways. If it is already in the current state, the goal is trivially achieved with no effort. Otherwise, we have to find some appropriate operator and try to apply it.
- An operator is appropriate if one of the effects of the operator is to add the goal in question to the current state; in other words, if the goal is in the operator's add-list.
- We can apply an operator if we can achieve all the preconditions. But this is easy, because we just defined the notion of achieving a goal in the previous paragraph. Once the preconditions have been achieved, applying an operator means executing the action and updating the current state in term of the operator's add-list and delete-list. Since our program is just a simulation—it won't be actually driving a car or dialing a telephone—we must be content simply to print out the action, rather than taking any real action.

## 4.3 Stage 3: Implementation

The specification is complete enough to lead directly to a complete Common Lisp program. Figure 4.1 summarizes the variables, data types, and functions that make up the GPS program, along with some of the Common Lisp functions used to implement it.

	<b>Top-Level Function</b>
GPS	Solve a goal from a state using a list of operators.
	<b>Special Variables</b>
*state*	The current state: a list of conditions.
*ops*	A list of available operators.
	<b>Data Types</b>
op	An operation with preconds, add-list and del-list.
	<b>Functions</b>
achieve	Achieve an individual goal.
appropriate-p	Decide if an operator is appropriate for a goal.
apply-op	Apply operator to current state.
	<b>Selected Common Lisp Functions</b>
member	Test if an element is a member of a list. (p. 78)
set-difference	All elements in one set but not the other.
union	All elements in either of two sets.
every	Test if every element of a list passes a test. (p. 62)
some	Test if any element of a list passes a test.
	<b>Previously Defined Functions</b>
find-all	A list of all matching elements. (p. 101)

Figure 4.1: Glossary for the GPS Program

Here is the complete GPS program itself:

```
(defvar *state* nil "The current state: a list of conditions.")

(defvar *ops* nil "A list of available operators.")

(defstruct op "An operation"
  (action nil) (preconds nil) (add-list nil) (del-list nil))

(defun GPS (*state* goals *ops*)
  "General Problem Solver: achieve all goals using *ops*."
  (if (every #'achieve goals) 'solved))

(defun achieve (goal)
  "A goal is achieved if it already holds,
  or if there is an appropriate op for it that is applicable."
  (or (member goal *state*)
      (some #'apply-op
            (find-all goal *ops* :test #'appropriate-p)))))

(defun appropriate-p (goal op)
  "An op is appropriate to a goal if it is in its add list."
  (member goal (op-add-list op)))
```

---

```
(defun apply-op (op)
  "Print a message and update *state* if op is applicable."
  (when (every #'achieve (op-preconds op))
    (print (list 'executing (op-action op)))
    (setf *state* (set-difference *state* (op-del-list op)))
    (setf *state* (union *state* (op-add-list op)))
    t))
```

We can see the program is made up of seven definitions. These correspond to the seven items in the specification above. In general, you shouldn't expect such a perfect fit between specification and implementation. There are two defvar forms, one defstruct, and four defun forms. These are the Common Lisp forms for defining variables, structures, and functions, respectively. They are the most common top-level forms in Lisp, but there is nothing magic about them; they are just special forms that have the side effect of adding new definitions to the Lisp environment.

The two defvar forms, repeated below, declare special variables named **\*state\*** and **\*ops\***, which can then be accessed from anywhere in the program.

```
(defvar *state* nil "The current state: a list of conditions.")
(defvar *ops* nil "A list of available operators.")
```

The defstruct form defines a structure called an op, which has slots called action, preconds, add-list, and del-list. Structures in Common Lisp are similar to structures in C, or records in Pascal. The defstruct automatically defines a constructor function, which is called make-op, and an access function for each slot of the structure. The access functions are called op-action, op-preconds, op-add-list, and op-del-list. The defstruct also defines a copier function, copy-op, a predicate, op-p, and setf definitions for changing each slot. None of those are used in the GPS program. Roughly speaking, it is as if the defstruct form

```
(defstruct op "An operation"
  (action nil) (preconds nil) (add-list nil) (del-list nil))
```

expanded into the following definitions:

```
(defun make-op (&key action preconds add-list del-list)
  (vector 'op action preconds add-list del-list))

(defun op-action (op) (elt op 1))
(defun op-preconds (op) (elt op 2))
(defun op-add-list (op) (elt op 3))
(defun op-del-list (op) (elt op 4))

(defun copy-op (op) (copy-seq op))
```

```
(defun op-p (op)
  (and (vectorp op) (eq (elt op 0) 'op)))
  (setf (documentation 'op 'structure) "An operation")
```

Next in the GPS program are four function definitions. The main function, GPS, is passed three arguments. The first is the current state of the world, the second the goal state, and the third a list of allowable operators. The body of the function says simply that if we can achieve every one of the goals we have been given, then the problem is solved. The unstated alternative is that otherwise, the problem is not solved.

The function `achieve` is given as an argument a single goal. The function succeeds if that goal is already true in the current state (in which case we don't have to do anything) or if we can apply an appropriate operator. This is accomplished by first building the list of appropriate operators and then testing each in turn until one can be applied. `achieve` calls `find-all`, which we defined on page 101. In this use, `find-all` returns a list of operators that match the current goal, according to the predicate `appropriate-p`.

The function `appropriate-p` tests if an operator is appropriate for achieving a goal. (It follows the Lisp naming convention that predicates end in `-p`.)

Finally, the function `apply-op` says that if we can achieve all the preconditions for an appropriate operator, then we can apply the operator. This involves printing a message to that effect and changing the state of the world by deleting what was in the `delete-list` and adding what was in the `add-list`. `apply-op` is also a predicate; it returns `t` only when the operator can be applied.

## 4.4 Stage 4: Test

This section will define a list of operators applicable to the “driving to nursery school” domain and will show how to pose and solve some problems in that domain. First, we need to construct the list of operators for the domain. The `defstruct` form for the type `op` automatically defines the function `make-op`, which can be used as follows:

```
(make-op :action 'drive-son-to-school
        :preconds '(son-at-home car-works)
        :add-list '(son-at-school)
        :del-list '(son-at-home))
```

This expression returns an operator whose action is the symbol `drive-son-to-school` and whose preconditions, add-list and delete-list are the specified lists. The intent

of this operator is that whenever the son is at home and the car works, `drive-son-to-school` can be applied, changing the state by deleting the fact that the son is at home, and adding the fact that he is at school.

It should be noted that using long hyphenated atoms like `son-at-home` is a useful approach only for very simple examples like this one. A better representation would break the atom into its components: perhaps `(at son home)`. The problem with the atom-based approach is one of combinatorics. If there are 10 predicates (such as `at`) and 10 people or objects, then there will be  $10 \times 10 \times 10 = 1000$  possible hyphenated atoms, but only 20 components. Clearly, it would be easier to describe the components. In this chapter we stick with the hyphenated atoms because it is simpler, and we do not need to describe the whole world. Subsequent chapters take knowledge representation more seriously.

With this operator as a model, we can define other operators corresponding to Newell and Simon's quote on page 109. There will be an operator for installing a battery, telling the repair shop the problem, and telephoning the shop. We can fill in the "and so on" by adding operators for looking up the shop's phone number and for giving the shop money:

```
(defparameter *school-ops*
  (list
    (make-op :action 'drive-son-to-school
      :preconds '(son-at-home car-works)
      :add-list '(son-at-school)
      :del-list '(son-at-home))
    (make-op :action 'shop-installs-battery
      :preconds '(car-needs-battery shop-knows-problem shop-has-money)
      :add-list '(car-works))
    (make-op :action 'tell-shop-problem
      :preconds '(in-communication-with-shop)
      :add-list '(shop-knows-problem))
    (make-op :action 'telephone-shop
      :preconds '(know-phone-number)
      :add-list '(in-communication-with-shop))
    (make-op :action 'look-up-number
      :preconds '(have-phone-book)
      :add-list '(know-phone-number))
    (make-op :action 'give-shop-money
      :preconds '(have-money)
      :add-list '(shop-has-money)
      :del-list '(have-money))))
```

The next step is to pose some problems to GPS and examine the solutions. Following are three sample problems. In each case, the goal is the same: to achieve the single condition `son-at-school`. The list of available operators is also the same in each

problem; the difference is in the initial state. Each of the three examples consists of the prompt, ">", which is printed by the Lisp system, followed by a call to GPS, "(gps ...)", which is typed by the user, then the output from the program, "(EXECUTING ...)", and finally the result of the function call, which can be either SOLVED or NIL.

```
> (gps '(son-at-home car-needs-battery have-money have-phone-book)
      '(son-at-school)
      *school-ops*)
(Executing LOOK-UP-NUMBER)
(Executing TELEPHONE-SHOP)
(Executing TELL-SHOP-PROBLEM)
(Executing GIVE-SHOP-MONEY)
(Executing SHOP-INSTALLS-BATTERY)
(Executing DRIVE-SON-TO-SCHOOL)
SOLVED

> (gps '(son-at-home car-needs-battery have-money)
      '(son-at-school)
      *school-ops*)
NIL

> (gps '(son-at-home car-works)
      '(son-at-school)
      *school-ops*)
(Executing DRIVE-SON-TO-SCHOOL)
SOLVED
```

In all three examples the goal is to have the son at school. The only operator that has `son-at-school` in its add-list is `drive-son-to-school`, so GPS selects that operator initially. Before it can execute the operator, GPS has to solve for the preconditions. In the first example, the program ends up working backward through the operators `shop-installs-battery`, `give-shop-money`, `tell-shop-problem`, and `telephone-shop` to `look-up-number`, which has no outstanding preconditions. Thus, the `look-up-number` action can be executed, and the program moves on to the other actions. As Aristotle said, "What is the last in the order of analysis seems to be first in the order of becoming."

The second example starts out exactly the same, but the `look-up-number` operator fails because its precondition, `have-phone-book`, cannot be achieved. Knowing the phone number is a precondition, directly or indirectly, of all the operators, so no action is taken and GPS returns NIL.

Finally, the third example is much more direct; the initial state specifies that the car works, so the driving operator can be applied immediately.

## 4.5 Stage 5: Analysis, or “We Lied about the G”

In the sections that follow, we examine the question of just how general this General Problem Solver is. The next four sections point out limitations of our version of GPS, and we will show how to correct these limitations in a second version of the program.

One might ask if “limitations” is just a euphemism for “bugs.” Are we “enhancing” the program, or are we “correcting” it? There are no clear answers on this point, because we never insisted on an unambiguous problem description or specification. AI programming is largely exploratory programming; the aim is often to discover more about the problem area rather than to meet a clearly defined specification. This is in contrast to a more traditional notion of programming, where the problem is completely specified before the first line of code is written.

## 4.6 The Running Around the Block Problem

Representing the operator “driving from home to school” is easy: the precondition and delete-list includes being at home, and the add-list includes being at school. But suppose we wanted to represent “running around the block.” There would be no net change of location, so does that mean there would be no add- or delete-list? If so, there would be no reason ever to apply the operator. Perhaps the add-list should contain something like “got some exercise” or “feel tired,” or something more general like “experience running around the block.” We will return to this question later.

## 4.7 The Clobbered Sibling Goal Problem

Consider the problem of not only getting the child to school but also having some money left over to use for the rest of the day. GPS can easily solve this problem from the following initial condition:

```
> (gps '(son-at-home have-money car-works)
      '(have-money son-at-school)
      *school-ops*)
(EXECUTING DRIVE-SON-TO-SCHOOL)
SOLVED
```

However, in the next example GPS incorrectly reports success, when in fact it has spent the money on the battery.

```
> (gps '(son-at-home car-needs-battery have-money have-phone-book)
      '(have-money son-at-school)
      *school-ops*)
(EXECUTING LOOK-UP-NUMBER)
(EXECUTING TELEPHONE-SHOP)
(EXECUTING TELL-SHOP-PROBLEM)
(EXECUTING GIVE-SHOP-MONEY)
(EXECUTING SHOP-INSTALLS-BATTERY)
(EXECUTING DRIVE-SON-TO-SCHOOL)
SOLVED
```

The “bug” is that GPS uses the expression `(every #'achieve goals)` to achieve a set of goals. If this expression returns true, it means that every one of the goals has been achieved in sequence, but it doesn’t mean they are all still true at the end. In other words, the goal `(have-money son-at-school)`, which we intended to mean “end up in a state where both `have-money` and `son-at-school` are true,” was interpreted by GPS to mean “first achieve `have-money`, and then achieve `son-at-school`.“ Sometimes achieving one goal can undo another, previously achieved goal. We will call this the “prerequisite clobbers sibling goal” problem.<sup>2</sup> That is, `have-money` and `son-at-school` are sibling goals, one of the prerequisites for the plan for `son-at-school` is `car-works`, and achieving that goal clobbers the `have-money` goal.

Modifying the program to recognize the “prerequisite clobbers sibling goal” problem is straightforward. First note that we call `(every #'achieve something)` twice within the program, so let’s replace those two forms with `(achieve-all something)`. We can then define `achieve-all` as follows:

```
(defun achieve-all (goals)
  "Try to achieve each goal, then make sure they still hold."
  (and (every #'achieve goals) (subsetp goals *state*)))
```

The Common Lisp function `subsetp` returns true if its first argument is a subset of its second. In `achieve-all`, it returns true if every one of the goals is still in the current state after achieving all the goals. This is just what we wanted to test.

The introduction of `achieve-all` prevents GPS from returning true when one of the goals gets clobbered, but it doesn’t force GPS to replan and try to recover from a clobbered goal. We won’t consider that possibility now, but we will take it up again in the section on the blocks world domain, which was Sussman’s primary example.

---

<sup>2</sup>Gerald Sussman, in his book *A Computer Model of Skill Acquisition*, uses the term “prerequisite clobbers brother goal” or PCBG. I prefer to be gender neutral, even at the risk of being labeled a historical revisionist.

## 4.8 The Leaping before You Look Problem

Another way to address the “prerequisite clobbers sibling goal” problem is just to be more careful about the order of goals in a goal list. If we want to get the kid to school and still have some money left, why not just specify the goal as (son-at-school have-money) rather than (have-money son-at-school)? Let’s see what happens when we try that:

```
> (gps '(son-at-home car-needs-battery have-money have-phone-book)
      '(son-at-school have-money)
      *school-ops*)
(EXECUTING LOOK-UP-NUMBER)
(EXECUTING TELEPHONE-SHOP)
(EXECUTING TELL-SHOP-PROBLEM)
(EXECUTING GIVE-SHOP-MONEY)
(EXECUTING SHOP-INSTALLS-BATTERY)
(EXECUTING DRIVE-SON-TO-SCHOOL)
NIL
```

GPS returns nil, reflecting the fact that the goal cannot be achieved, but only after executing all actions up to and including driving to school. I call this the “leaping before you look” problem, because if you asked the program to solve for the two goals (jump-off-cliff land-safely) it would happily jump first, only to discover that it had no operator to land safely. This is less than prudent behavior.

The problem arises because planning and execution are interleaved. Once the preconditions for an operator are achieved, the action is taken—and **\*state\*** is irreversibly changed—even if this action may eventually lead to a dead end. An alternative would be to replace the single global **\*state\*** with distinct local state variables, such that a new variable is created for each new state. This alternative is a good one for another, independent reason, as we shall see in the next section.

## 4.9 The Recursive Subgoal Problem

In our simulated nursery school world there is only one way to find out a phone number: to look it up in the phone book. Suppose we want to add an operator for finding out a phone number by asking someone. Of course, in order to ask someone something, you need to be in communication with him or her. The asking-for-a-phone-number operator could be implemented as follows:

```
(push (make-op :action 'ask-phone-number
               :preconds '(in-communication-with-shop)
               :add-list '(know-phone-number))
      *school-ops*)
```

(The special form (*push item list*) puts the item on the front of the list; it is equivalent to (*setf list (cons item list)*) in the simple case.) Unfortunately, something unexpected happens when we attempt to solve seemingly simple problems with this new set of operators. Consider the following:

```
> (gps '(son-at-home car-needs-battery have-money)
      '(son-at-school)
      *school-ops*)

>>TRAP 14877 (SYSTEM:PDL-OVERFLOW EH::REGULAR)
The regular push-down list has overflowed.
While in the function ACHIEVE <- EVERY <- REMOVE
```

The error message (which will vary from one implementation of Common Lisp to another) means that too many recursively nested function calls were made. This indicates either a very complex problem or, more commonly, a bug in the program leading to infinite recursion. One way to try to see the cause of the bug is to trace a relevant function, such as *achieve*:

```
> (trace achieve) => (ACHIEVE)

> (gps '(son-at-home car-needs-battery have-money)
      '(son-at-school)
      *school-ops*)
(1 ENTER ACHIEVE: SON-AT-SCHOOL)
(2 ENTER ACHIEVE: SON-AT-HOME)
(2 EXIT ACHIEVE: (SON-AT-HOME CAR-NEEDS-BATTERY HAVE-MONEY))
(2 ENTER ACHIEVE: CAR-WORKS)
(3 ENTER ACHIEVE: CAR-NEEDS-BATTERY)
(3 EXIT ACHIEVE: (CAR-NEEDS-BATTERY HAVE-MONEY))
(3 ENTER ACHIEVE: SHOP-KNOWS-PROBLEM)
(4 ENTER ACHIEVE: IN-COMMUNICATION-WITH-SHOP)
(5 ENTER ACHIEVE: KNOW-PHONE-NUMBER)
(6 ENTER ACHIEVE: IN-COMMUNICATION-WITH-SHOP)
(7 ENTER ACHIEVE: KNOW-PHONE-NUMBER)
(8 ENTER ACHIEVE: IN-COMMUNICATION-WITH-SHOP)
(9 ENTER ACHIEVE: KNOW-PHONE-NUMBER)
```

The output from trace gives us the necessary clues. Newell and Simon talk of “oscillating among ends, functions required, and means that perform them.” Here it seems we have an infinite oscillation between being in communication with the shop (levels 4, 6, 8, ...) and knowing the shop’s phone number (levels 5, 7, 9, ...). The reasoning is as follows: we want the shop to know about the problem with the battery, and this requires being in communication with him or her. One way to get in communication is to phone, but we don’t have a phone book to look up the number. We could ask them their phone number, but this requires being in communication with them. As Aristotle put it, “If we are to be always deliberating, we shall have to go on to infinity.” We will call this the “recursive subgoal” problem: trying to solve a problem in terms of itself. One way to avoid the problem is to have achieve keep track of all the goals that are being worked on and give up if it sees a loop in the goal stack.

## 4.10 The Lack of Intermediate Information Problem

When GPS fails to find a solution, it just returns nil. This is annoying in cases where the user expected a solution to be found, because it gives no information about the cause of failure. The user could always trace some function, as we traced achieve above, but the output from trace is rarely exactly the information desired. It would be nice to have a general debugging output tool where the programmer could insert print statements into his code and have them selectively printed, depending on the information desired.

The function dbg provides this capability. dbg prints output in the same way as format, but it will only print when debugging output is desired. Each call to dbg is accompanied by an identifier that is used to specify a class of debugging messages. The functions debug and undebug are used to add or remove message classes to the list of classes that should be printed. In this chapter, all the debugging output will use the identifier :gps. Other programs will use other identifiers, and a complex program will use many identifiers.

A call to dbg will result in output if the first argument to dbg, the identifier, is one that was specified in a call to debug. The other arguments to dbg are a format string followed by a list of arguments to be printed according to the format string. In other words, we will write functions that include calls to dbg like:

```
(dbg :gps "The current goal is: ~a" goal)
```

If we have turned on debugging with (debug :gps), then calls to dbg with the identifier :gps will print output. The output is turned off with (undebug :gps).

debug and undebug are designed to be similar to trace and untrace, in that they turn diagnostic output on and off. They also follow the convention that debug with no arguments returns the current list of identifiers, and that undebug with no arguments turns all debugging off. However, they differ from trace and untrace in that they are functions, not macros. If you use only keywords and integers for identifiers, then you won't notice the difference.

Two new built-in features are introduced here. First, \*debug-io\* is the stream normally used for debugging input/output. In all previous calls to format we have used t as the stream argument, which causes output to go to the \*standard-output\* stream. Sending different types of output to different streams allows the user some flexibility. For example, debugging output could be directed to a separate window, or it could be copied to a file. Second, the function fresh-line advances to the next line of output, unless the output stream is already at the start of the line.

```
(defvar *dbg-ids* nil "Identifiers used by dbg")

(defun dbg (id format-string &rest args)
  "Print debugging info if (DEBUG ID) has been specified."
  (when (member id *dbg-ids*)
    (fresh-line *debug-io*)
    (apply #'format *debug-io* format-string args)))

(defun debug (&rest ids)
  "Start dbg output on the given ids."
  (setf *dbg-ids* (union ids *dbg-ids*)))

(defun undebug (&rest ids)
  "Stop dbg on the ids. With no ids, stop dbg altogether."
  (setf *dbg-ids* (if (null ids) nil
                      (set-difference *dbg-ids* ids))))
```

Sometimes it is easier to view debugging output if it is indented according to some pattern, such as the depth of nested calls to a function. To generate indented output, the function dbg-indent is defined:

```
(defun dbg-indent (id indent format-string &rest args)
  "Print indented debugging info if (DEBUG ID) has been specified."
  (when (member id *dbg-ids*)
    (fresh-line *debug-io*)
    (dotimes (i indent) (princ " " *debug-io*))
    (apply #'format *debug-io* format-string args)))
```

## 4.11 GPS Version 2: A More General Problem Solver

At this point we are ready to put together a new version of GPS with solutions for the “running around the block,” “prerequisite clobbers sibling goal,” “leaping before you look,” and “recursive subgoal” problems. The glossary for the new version is in figure 4.2.

<b>Top-Level Function</b>	
GPS	Solve a goal from a state using a list of operators.
<b>Special Variables</b>	
*ops*	A list of available operators.
<b>Data Types</b>	
op	An operation with preconds, add-list and del-list.
<b>Major Functions</b>	
achieve-all	Achieve a list of goals.
achieve	Achieve an individual goal.
appropriate-p	Decide if an operator is appropriate for a goal.
apply-op	Apply operator to current state.
<b>Auxiliary Functions</b>	
executing-p	Is a condition an executing form?
starts-with	Is the argument a list that starts with a given atom?
convert-op	Convert an operator to use the executing convention.
op	Create an operator.
use	Use a list of operators.
member-equal	Test if an element is equal to a member of a list.
<b>Selected Common Lisp Functions</b>	
member	Test if an element is a member of a list. (p. 78)
set-difference	All elements in one set but not the other.
subsetp	Is one set wholly contained in another?
union	All elements in either of two sets.
every	Test if every element of a list passes a test. (p. 62)
some	Test if any element of a list passes a test.
remove-if	Remove all items satisfying a test.
<b>Previously Defined Functions</b>	
find-all	A list of all matching elements. (p. 101)
find-all-if	A list of all elements satisfying a predicate.

Figure 4.2: Glossary for Version 2 of GPS

The most important change is that, instead of printing a message when each operator is applied, we will instead have GPS return the resulting state. A list of

“messages” in each state indicates what actions have been taken. Each message is actually a condition, a list of the form (*executing operator*). This solves the “running around the block” problem: we could call GPS with an initial goal of ((*executing run-around-block*)), and it would execute the *run-around-block* operator, thereby satisfying the goal. The following code defines a new function, *op*, which builds operators that include the message in their add-list.

```
(defun executing-p (x)
  "Is x of the form: (executing ...) ?"
  (starts-with x 'executing))

(defun starts-with (list x)
  "Is this a list whose first element is x?"
  (and (consp list) (eq1 (first list) x)))

(defun convert-op (op)
  "Make op conform to the (EXECUTING op) convention."
  (unless (some #'executing-p (op-add-list op))
    (push (list 'executing (op-action op)) (op-add-list op)))
  op)

(defun op (action &key preconds add-list del-list)
  "Make a new operator that obeys the (EXECUTING op) convention."
  (convert-op
    (make-op :action action :preconds preconds
      :add-list add-list :del-list del-list)))
```

Operators built by *op* will be correct, but we can convert existing operators using *convert-op* directly:

```
(mapc #'convert-op *school-ops*)
```

This is an example of exploratory programming: instead of starting all over when we discover a limitation of the first version, we can use Lisp to alter existing data structures for the new version of the program.

The definition of the variable *\*ops\** and the structure *op* are exactly the same as before, and the rest of the program consists of five functions we have already seen: GPS, achieve-all, achieve, appropriate-p, and apply-op. At the top level, the function GPS calls achieve-all, which returns either nil or a valid state. From this we remove all the atoms, which leaves only the elements of the final state that are lists—in other words, the actions of the form (*executing operator*). Thus, the value of GPS itself is the list of actions taken to arrive at the final state. GPS no longer returns SOLVED when it finds a solution, but it still obeys the convention of returning nil for failure, and non-nil for success. In general, it is a good idea to have a program return

a meaningful value rather than print that value, if there is the possibility that some other program might ever want to use the value.

```
(defvar *ops* nil "A list of available operators.")

(defstruct op "An operation"
  (action nil) (preconds nil) (add-list nil) (del-list nil))

(defun GPS (state goals &optional (*ops* *ops*))
  "General Problem Solver: from state, achieve goals using *ops*."
  (remove-if #'atom (achieve-all (cons '(start) state) goals nil)))
```

The first major change in version 2 is evident from the first line of the program: there is no `*state*` variable. Instead, the program keeps track of local state variables. This is to solve the “leaping before you look” problem, as outlined before. The functions `achieve`, `achieve-all`, and `apply-op` all take an extra argument which is the current state, and all return a new state as their value. They also must still obey the convention of returning nil when they fail.

Thus we have a potential ambiguity: does nil represent failure, or does it represent a valid state that happens to have no conditions? We resolve the ambiguity by adopting the convention that all states must have at least one condition. This convention is enforced by the function `GPS`. Instead of calling `(achieve-all state goals nil)`, `GPS` calls `(achieve-all (cons '(start) state) goals nil)`. So even if the user passes `GPS` a null initial state, it will pass on a state containing `(start)` to `achieve-all`. From then on, we are guaranteed that no state will ever become nil, because the only function that builds a new state is `apply-op`, and we can see by looking at the last line of `apply-op` that it always appends something onto the state it is returning. (An add-list can never be nil, because if it were, the operator would not be appropriate. Besides, every operator includes the `(executing ...)` condition.)

Note that the final value we return from `GPS` has all the atoms removed, so we end up reporting only the actions performed, since they are represented by conditions of the form `(executing action)`. Adding the `(start)` condition at the beginning also serves to differentiate between a problem that cannot be solved and one that is solved without executing any actions. Failure returns nil, while a solution with no steps will at least include the `(start)` condition, if nothing else.

Functions that return nil as an indication of failure and return some useful value otherwise are known as *semipredicates*. They are error prone in just these cases where nil might be construed as a useful value. Be careful when defining and using semipredicates: (1) Decide if nil could ever be a meaningful value. (2) Insure that the *user* can’t corrupt the program by supplying nil as a value. In this program, `GPS` is the only function the user should call, so once we have accounted for it, we’re covered. (3) Insure that the *program* can’t supply nil as a value. We did this by seeing that there was only one place in the program where new states were constructed, and that this new state was formed by appending a one-element list onto another

state. By following this three-step procedure, we have an informal proof that the semipredicates involving states will function properly. This kind of informal proof procedure is a common element of good program design.

The other big change in version 2 is the introduction of a goal stack to solve the recursive subgoal problem. The program keeps track of the goals it is working on and immediately fails if a goal appears as a subgoal of itself. This test is made in the second clause of achieve.

The function achieve-all tries to achieve each one of the goals in turn, setting the variable state2 to be the value returned from each successive call to achieve. If all goals are achieved in turn, and if all the goals still hold at the end (as subsetp checks for), then the final state is returned; otherwise the function fails, returning nil.

Most of the work is done by achieve, which gets passed a state, a single goal condition, and the stack of goals worked on so far. If the condition is already in the state, then achieve succeeds and returns the state. On the other hand, if the goal condition is already in the goal stack, then there is no sense continuing—we will be stuck in an endless loop—so achieve returns nil. Otherwise, achieve looks through the list of operators, trying to find one appropriate to apply.

```
(defun achieve-all (state goals goal-stack)
  "Achieve each goal, and make sure they still hold at the end."
  (let ((current-state state))
    (if (and (every #'(lambda (g)
                         (setf current-state
                               (achieve current-state g goal-stack)))
                     goals)
              (subsetp goals current-state :test #'equal))
        current-state)
        (subsetp goals current-state :test #'equal)
        current-state)))

(defun achieve (state goal goal-stack)
  "A goal is achieved if it already holds,
  or if there is an appropriate op for it that is applicable."
  (dbg-indent :gps (length goal-stack) "Goal: ~a" goal)
  (cond ((member-equal goal state) state)
        ((member-equal goal goal-stack) nil)
        (t (some #'(lambda (op) (apply-op state goal op goal-stack))
                  (find-all goal *ops* :test #'appropriate-p)))))
```

The goal ((executing run-around-block)) is a list of one condition, where the condition happens to be a two-element list. Allowing lists as conditions gives us more flexibility, but we also have to be careful. The problem is that not all lists that look alike actually are the same. The predicate equal essentially tests to see if its two arguments look alike, while the predicate eql tests to see if its two arguments actually are identical. Since functions like member use eql by default, we have to specify with a :test keyword that we want equal instead. Since this is done several times, we

introduce the function `member-equal`. In fact, we could have carried the abstraction one step further and defined `member-situation`, a function to test if a condition is true in a situation. This would allow the user to change the matching function from `eql` to `equal`, and to anything else that might be useful.

```
(defun member-equal (item list)
  (member item list :test #'equal))
```

The function `apply-op`, which used to change the state irrevocably and print a message reflecting this, now returns the new state instead of printing anything. It first computes the state that would result from achieving all the preconditions of the operator. If it is possible to arrive at such a state, then `apply-op` returns a new state derived from this state by adding what's in the add-list and removing everything in the delete-list.

```
(defun apply-op (state goal op goal-stack)
  "Return a new, transformed state if op is applicable."
  (dbg-indent :gps (length goal-stack) "Consider: ~a" (op-action op))
  (let ((state2 (achieve-all state (op-preconds op)
                             (cons goal goal-stack))))
    (unless (null state2)
      ;; Return an updated state
      (dbg-indent :gps (length goal-stack) "Action: ~a" (op-action op))
      (append (remove-if #'(lambda (x)
                             (member-equal x (op-del-list op)))
                           state2)
              (op-add-list op)))))

(defun appropriate-p (goal op)
  "An op is appropriate to a goal if it is in its add-list."
  (member-equal goal (op-add-list op)))
```

There is one last complication in the way we compute the new state. In version 1 of GPS, states were (conceptually) unordered sets of conditions, so we could use `union` and `set-difference` to operate on them. In version 2, states become ordered lists, because we need to preserve the ordering of actions. Thus, we have to use the functions `append` and `remove-if`, since these are defined to preserve order, while `union` and `set-difference` are not.

Finally, the last difference in version 2 is that it introduces a new function: `use`. This function is intended to be used as a sort of declaration that a given list of operators is to be used for a series of problems.

```
(defun use (oplist)
  "Use oplist as the default list of operators."
  #: Return something useful, but not too verbose:
  #: the number of operators.
  (length (setf *ops* oplist)))
```

Calling `use` sets the parameter `*ops*`, so that it need not be specified on each call to GPS. Accordingly, in the definition of GPS itself the third argument, `*ops*`, is now optional; if it is not supplied, a default will be used. The default value for `*ops*` is given as `*ops*`. This may seem redundant or superfluous—how could a variable be its own default? The answer is that the two occurrences of `*ops*` look alike, but they actually refer to two completely separate bindings of the special variable `*ops*`. Most of the time, variables in parameter lists are local variables, but there is no rule against binding a special variable as a parameter. Remember that the effect of binding a special variable is that all references to the special variable that occur anywhere in the program—even outside the lexical scope of the function—refer to the new binding of the special variable. So after a sequence of calls we eventually reach `achieve`, which references `*ops*`, and it will see the newly bound value of `*ops*`.

The definition of GPS is repeated here, along with an alternate version that binds a local variable and explicitly sets and resets the special variable `*ops*`. Clearly, the idiom of binding a special variable is more concise, and while it can be initially confusing, it is useful once understood.

```
(defun GPS (state goals &optional (*ops* *ops*))
  "General Problem Solver: from state, achieve goals using *ops*."
  (remove-if #'atom (achieve-all (cons '(start) state) goals nil)))

(defun GPS (state goals &optional (ops *ops*))
  "General Problem Solver: from state, achieve goals using *ops*."
  (let ((old-ops *ops*))
    (setf *ops* ops)
    (let ((result (remove-if #'atom (achieve-all
                                     (cons '(start) state)
                                     goals nil))))
      (setf *ops* old-ops)
      result)))
```

Now let's see how version 2 performs. We use the list of operators that includes the “asking the shop their phone number” operator. First we make sure it will still do the examples version 1 did:

```
> (use *school-ops*) ⇒ 7
```

```
> (gps '(son-at-home car-needs-battery have-money have-phone-book)
      '(son-at-school))
((START)
 (EXECUTING LOOK-UP-NUMBER)
 (EXECUTING TELEPHONE-SHOP)
 (EXECUTING TELL-SHOP-PROBLEM)
 (EXECUTING GIVE-SHOP-MONEY)
 (EXECUTING SHOP-INSTALLS-BATTERY)
 (EXECUTING DRIVE-SON-TO-SCHOOL))

> (debug :gps) => (:GPS)

> (gps '(son-at-home car-needs-battery have-money have-phone-book)
      '(son-at-school))
Goal: SON-AT-SCHOOL
Consider: DRIVE-SON-TO-SCHOOL
Goal: SON-AT-HOME
Goal: CAR-WORKS
Consider: SHOP-INSTALLS-BATTERY
Goal: CAR-NEEDS-BATTERY
Goal: SHOP-KNOWS-PROBLEM
Consider: TELL-SHOP-PROBLEM
Goal: IN-COMMUNICATION-WITH-SHOP
Consider: TELEPHONE-SHOP
Goal: KNOW-PHONE-NUMBER
Consider: ASK-PHONE-NUMBER
Goal: IN-COMMUNICATION-WITH-SHOP
Consider: LOOK-UP-NUMBER
Goal: HAVE-PHONE-BOOK
Action: LOOK-UP-NUMBER
Action: TELEPHONE-SHOP
Action: TELL-SHOP-PROBLEM
Goal: SHOP-HAS-MONEY
Consider: GIVE-SHOP-MONEY
Goal: HAVE-MONEY
Action: GIVE-SHOP-MONEY
Action: SHOP-INSTALLS-BATTERY
Action: DRIVE-SON-TO-SCHOOL
((START)
 (EXECUTING LOOK-UP-NUMBER)
 (EXECUTING TELEPHONE-SHOP)
 (EXECUTING TELL-SHOP-PROBLEM)
 (EXECUTING GIVE-SHOP-MONEY)
 (EXECUTING SHOP-INSTALLS-BATTERY)
 (EXECUTING DRIVE-SON-TO-SCHOOL))

> (udebug) => NIL
```

```
> (gps '(son-at-home car-works)
      '(son-at-school))
((START)
 (EXECUTING DRIVE-SON-TO-SCHOOL))
```

Now we see that version 2 can also handle the three cases that version 1 got wrong. In each case, the program avoids an infinite loop, and also avoids leaping before it looks.

```
> (gps '(son-at-home car-needs-battery have-money have-phone-book)
      '(have-money son-at-school))
NIL

> (gps '(son-at-home car-needs-battery have-money have-phone-book)
      '(son-at-school have-money))
NIL

> (gps '(son-at-home car-needs-battery have-money)
      '(son-at-school))
NIL
```

Finally, we see that this version of GPS also works on trivial problems requiring no action:

```
> (gps '(son-at-home) '(son-at-home)) => ((START))
```

## 4.12 The New Domain Problem: Monkey and Bananas

To show that GPS is at all general, we have to make it work in different domains. We will start with a “classic” AI problem.<sup>3</sup> Imagine the following scenario: a hungry monkey is standing at the doorway to a room. In the middle of the room is a bunch of bananas suspended from the ceiling by a rope, well out of the monkey’s reach. There is a chair near the door, which is light enough for the monkey to push and tall enough to reach almost to the bananas. Just to make things complicated, assume the monkey is holding a toy ball and can only hold one thing at a time.

In trying to represent this scenario, we have some flexibility in choosing what to put in the current state and what to put in with the operators. For now, assume we define the operators as follows:

---

<sup>3</sup>Originally posed by Saul Amarel (1968).

```
(defparameter *banana-ops*
  (list
    (op 'climb-on-chair
        :preconds '(chair-at-middle-room at-middle-room on-floor)
        :add-list '(at-bananas on-chair)
        :del-list '(at-middle-room on-floor))
    (op 'push-chair-from-door-to-middle-room
        :preconds '(chair-at-door at-door)
        :add-list '(chair-at-middle-room at-middle-room)
        :del-list '(chair-at-door at-door))
    (op 'walk-from-door-to-middle-room
        :preconds '(at-door on-floor)
        :add-list '(at-middle-room)
        :del-list '(at-door))
    (op 'grasp-bananas
        :preconds '(at-bananas empty-handed)
        :add-list '(has-bananas)
        :del-list '(empty-handed))
    (op 'drop-ball
        :preconds '(has-ball)
        :add-list '(empty-handed)
        :del-list '(has-ball))
    (op 'eat-bananas
        :preconds '(has-bananas)
        :add-list '(empty-handed not-hungry)
        :del-list '(has-bananas hungry))))
```

Using these operators, we could pose the problem of becoming not-hungry, given the initial state of being at the door, standing on the floor, holding the ball, hungry, and with the chair at the door. GPS can find a solution to this problem:

```
> (use *banana-ops*) => 6
> (GPS '(at-door on-floor has-ball hungry chair-at-door)
      '(not-hungry))
((START)
 (EXECUTING PUSH-CHAIR-FROM-DOOR-TO-MIDDLE-ROOM)
 (EXECUTING CLIMB-ON-CHAIR)
 (EXECUTING DROP-BALL)
 (EXECUTING GRASP-BANANAS)
 (EXECUTING EAT-BANANAS))
```

Notice we did not need to make any changes at all to the GPS program. We just used a different set of operators.

## 4.13 The Maze Searching Domain

Now we will consider another “classic” problem, maze searching. We will assume a particular maze, diagrammed here.

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25

It is much easier to define some functions to help build the operators for this domain than it would be to type in all the operators directly. The following code defines a set of operators for mazes in general, and for this maze in particular:

```
(defun make-maze-ops (pair)
  "Make maze ops in both directions"
  (list (make-maze-op (first pair) (second pair))
        (make-maze-op (second pair) (first pair)))))

(defun make-maze-op (here there)
  "Make an operator to move between two places"
  (op '(move from ,here to ,there)
       :preconds '((at ,here))
       :add-list '((at ,there))
       :del-list '((at ,here)))))

(defparameter *maze-ops*
  (mappend #'make-maze-ops
    '(((1 2) (2 3) (3 4) (4 9) (9 14) (9 8) (8 7) (7 12) (12 13)
      (12 11) (11 6) (11 16) (16 17) (17 22) (21 22) (22 23)
      (23 18) (23 24) (24 19) (19 20) (20 15) (15 10) (10 5) (20 25))))
```

Note the backquote notation, (`). It is covered in section 3.2, page 67.

We can now use this list of operators to solve several problems with this maze. And we could easily create another maze by giving another list of connections. Note that there is nothing that says the places in the maze are arranged in a five-by-five layout—that is just one way of visualizing the connectivity.

```
> (use *maze-ops*) ⇒ 48
```

```
> (gps '((at 1)) '((at 25)))
((START)
 (EXECUTING (MOVE FROM 1 TO 2))
 (EXECUTING (MOVE FROM 2 TO 3))
 (EXECUTING (MOVE FROM 3 TO 4))
 (EXECUTING (MOVE FROM 4 TO 9))
 (EXECUTING (MOVE FROM 9 TO 8))
 (EXECUTING (MOVE FROM 8 TO 7))
 (EXECUTING (MOVE FROM 7 TO 12))
 (EXECUTING (MOVE FROM 12 TO 11))
 (EXECUTING (MOVE FROM 11 TO 16))
 (EXECUTING (MOVE FROM 16 TO 17))
 (EXECUTING (MOVE FROM 17 TO 22))
 (EXECUTING (MOVE FROM 22 TO 23))
 (EXECUTING (MOVE FROM 23 TO 24))
 (EXECUTING (MOVE FROM 24 TO 19))
 (EXECUTING (MOVE FROM 19 TO 20))
 (EXECUTING (MOVE FROM 20 TO 25))
(AT 25))
```

There is one subtle bug that the maze domain points out. We wanted GPS to return a list of the actions executed. However, in order to account for the case where the goal can be achieved with no action, I included (START) in the value returned by GPS. These examples include the START and EXECUTING forms but also a list of the form (AT *n*), for some *n*. This is the bug. If we go back and look at the function GPS, we find that it reports the result by removing all atoms from the state returned by achieve-all. This is a “pun”—we said remove atoms, when we really meant to remove all conditions except the (START) and (EXECUTING *action*) forms. Up to now, all these conditions were atoms, so this approach worked. The maze domain introduced conditions of the form (AT *n*), so for the first time there was a problem. The moral is that when a programmer uses puns—saying what’s convenient instead of what’s really happening—there’s bound to be trouble. What we really want to do is not to remove atoms but to find all elements that denote actions. The code below says what we mean:

```
(defun GPS (state goals &optional (*ops* *ops*))
  "General Problem Solver: from state, achieve goals using *ops*."
  (find-all-if #'action-p
    (achieve-all (cons '(start) state) goals nil)))
```

---

```
(defun action-p (x)
  "Is x something that is (start) or (executing ...)?"
  (or (equal x '(start)) (executing-p x)))
```

The domain of maze solving also points out an advantage of version 2: that it returns a representation of the actions taken rather than just printing them out. The reason this is an advantage is that we may want to use the results for something, rather than just look at them. Suppose we wanted a function that gives us a path through a maze as a list of locations to visit in turn. We could do this by calling GPS as a subfunction and then manipulating the results:

```
(defun find-path (start end)
  "Search a maze for a path from start to end."
  (let ((results (GPS '((at ,start) '((at ,end)))))
        (unless (null results)
          (cons start (mapcar #'destination
                               (remove '(start) results
                                       :test #'equal))))))

  (defun destination (action)
    "Find the Y in (executing (move from X to Y))"
    (fifth (second action)))
```

The function `find-path` calls GPS to get the results. If this is `nil`, there is no answer, but if it is not, then take the rest of `results` (in other words, ignore the (`START`) part). Pick out the destination, `y`, from each (`EXECUTING (MOVE FROM x TO y)`) form, and remember to include the starting point.

```
> (use *maze-ops*) ⇒ 48
> (find-path 1 25) ⇒
(1 2 3 4 9 8 7 12 11 16 17 22 23 24 19 20 25)
> (find-path 1 1) ⇒ (1)
> (equal (find-path 1 25) (reverse (find-path 25 1))) ⇒ T
```

## 4.14 The Blocks World Domain

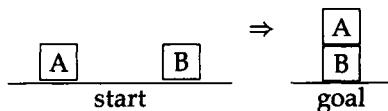
Another domain that has attracted more than its share of attention in AI circles is the blocks world domain. Imagine a child's set of building blocks on a table top. The problem is to move the blocks from their starting configuration into some goal configuration. We will assume that each block can have only one other block directly

on top of it, although they can be stacked to arbitrary height. The only action that can be taken in this world is to move a single block that has nothing on top of it either to the top of another block or onto the table that represents the block world. We will create an operator for each possible block move.

```
(defun make-block-ops (blocks)
  (let ((ops nil))
    (dolist (a blocks)
      (dolist (b blocks)
        (unless (equal a b)
          (dolist (c blocks)
            (unless (or (equal c a) (equal c b))
              (push (move-op a b c) ops)))
            (push (move-op a 'table b) ops)
            (push (move-op a b 'table) ops))))
        ops)))
  (defun move-op (a b c)
    "Make an operator to move A from B to C."
    (op '(move ,a from ,b to ,c)
        :preconds '((space on ,a) (space on ,c) (,a on ,b))
        :add-list (move-ons a b c)
        :del-list (move-ons a c b)))

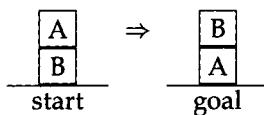
  (defun move-ons (a b c)
    (if (eq b 'table)
        '((,a on ,c))
        '((,a on ,c) (space on ,b))))
```

Now we try these operators out on some problems. The simplest possible problem is stacking one block on another:



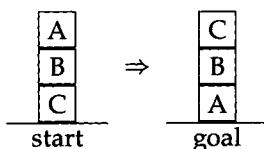
```
> (use (make-block-ops '(a b))) ⇒ 4
> (gps '((a on table) (b on table) (space on a) (space on b)
         (space on table))
       '((a on b) (b on table)))
((START)
 (EXECUTING (MOVE A FROM TABLE TO B)))
```

Here is a slightly more complex problem: inverting a stack of two blocks. This time we show the debugging output.



```
> (debug :gps) => (:GPS)
> (gps '((a on b) (b on table) (space on a) (space on table))
  '((b on a)))
Goal: (B ON A)
Consider: (MOVE B FROM TABLE TO A)
  Goal: (SPACE ON B)
  Consider: (MOVE A FROM B TO TABLE)
    Goal: (SPACE ON A)
    Goal: (SPACE ON TABLE)
    Goal: (A ON B)
  Action: (MOVE A FROM B TO TABLE)
  Goal: (SPACE ON A)
  Goal: (B ON TABLE)
Action: (MOVE B FROM TABLE TO A)
((START)
 (EXECUTING (MOVE A FROM B TO TABLE))
 (EXECUTING (MOVE B FROM TABLE TO A)))
> (udebug) => NIL
```

Sometimes it matters what order you try the conjuncts in. For example, you can't have your cake and eat it too, but you can take a picture of your cake and eat it too, as long as you take the picture *before* eating it. In the blocks world, we have:



```
> (use (make-block-ops '(a b c))) => 18
> (gps '((a on b) (b on c) (c on table) (space on a) (space on table))
  '((b on a) (c on b)))
((START)
 (EXECUTING (MOVE A FROM B TO TABLE))
 (EXECUTING (MOVE B FROM C TO A))
 (EXECUTING (MOVE C FROM TABLE TO B)))
```

```
> (gps '((a on b) (b on c) (c on table) (space on a) (space on table))
      '((c on b) (b on a)))
NIL
```

In the first case, the tower was built by putting B on A first, and then C on B. In the second case, the program gets C on B first, but clobbers that goal while getting B on A. The “prerequisite clobbers sibling goal” situation is recognized, but the program doesn’t do anything about it. One thing we could do is try to vary the order of the conjunct goals. That is, we could change achieve-all as follows:

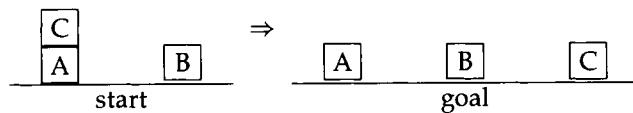
```
(defun achieve-all (state goals goal-stack)
  "Achieve each goal, trying several orderings."
  (some #'(lambda (goals) (achieve-each state goals goal-stack))
        (orderings goals)))

(defun achieve-each (state goals goal-stack)
  "Achieve each goal, and make sure they still hold at the end."
  (let ((current-state state))
    (if (and (every #'(lambda (g)
                         (setf current-state
                               (achieve current-state g goal-stack)))
                      goals)
              (subsetp goals current-state :test #'equal))
        current-state)))

(defun orderings (l)
  (if (> (length l) 1)
      (list l (reverse l))
      (list l)))
```

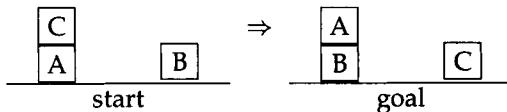
Now we can represent the goal either way, and we’ll still get an answer. Notice that we only consider two orderings: the order given and the reversed order. Obviously, for goal sets of one or two conjuncts this is all the orderings. In general, if there is only one interaction per goal set, then one of these two orders will work. Thus, we are assuming that “prerequisite clobbers sibling goal” interactions are rare, and that there will seldom be more than one interaction per goal set. Another possibility would be to consider all possible permutations of the goals, but that could take a long time with large goal sets.

Another consideration is the efficiency of solutions. Consider the simple task of getting block C on the table in the following diagram:



```
> (gps '((c on a) (a on table) (b on table)
          (space on c) (space on b) (space on table))
         '((c on table)))
((START)
 (EXECUTING (MOVE C FROM A TO B))
 (EXECUTING (MOVE C FROM B TO TABLE)))
```

The solution is correct, but there is an easier solution that moves C directly to the table. The simpler solution was not found because of an accident: it happens that make-block-ops defines the operators so that moving C from B to the table comes before moving C from A to the table. So the first operator is tried, and it succeeds provided C is on B. Thus, the two-step solution is found before the one-step solution is ever considered. The following example takes four steps when it could be done in two:



```
> (gps '((c on a) (a on table) (b on table)
          (space on c) (space on b) (space on table))
         '((c on table) (a on b)))
((START)
 (EXECUTING (MOVE C FROM A TO B))
 (EXECUTING (MOVE C FROM B TO TABLE))
 (EXECUTING (MOVE A FROM TABLE TO C))
 (EXECUTING (MOVE A FROM C TO B)))
```

How could we find shorter solutions? One way would be to do a full-fledged search: shorter solutions are tried first, temporarily abandoned when something else looks more promising, and then reconsidered later on. This approach is taken up in chapter 6, using a general searching function. A less drastic solution is to do a limited rearrangement of the order in which operators are searched: the ones with fewer unfulfilled preconditions are tried first. In particular, this means that operators with all preconditions filled would always be tried before other operators. To implement this approach, we change achieve:

```
(defun achieve (state goal goal-stack)
  "A goal is achieved if it already holds,
  or if there is an appropriate op for it that is applicable."
  (dbg-indent :gps (length goal-stack) "Goal: ~a" goal)
  (cond ((member-equal goal state) state)
        ((member-equal goal goal-stack) nil)))
```

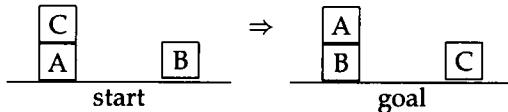
```

(t (some #'(lambda (op) (apply-op state goal op goal-stack))
            (appropriate-ops goal state)))) ;***

(defun appropriate-ops (goal state)
  "Return a list of appropriate operators,
  sorted by the number of unfulfilled preconditions."
  (sort (copy-list (find-all goal *ops* :test #'appropriate-p)) #<
        :key #'(lambda (op)
                  (count-if #'(lambda (precond)
                                (not (member-equal precond state)))
                            (op-preconds op)))))

```

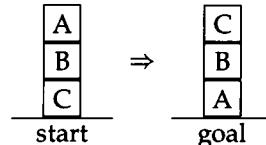
Now we get the solutions we wanted:



```

> (gps '((c on a) (a on table) (b on table)
          (space on c) (space on b) (space on table))
         '((c on table) (a on b)))
((START)
 (EXECUTING (MOVE C FROM A TO TABLE))
 (EXECUTING (MOVE A FROM TABLE TO B)))

```



```

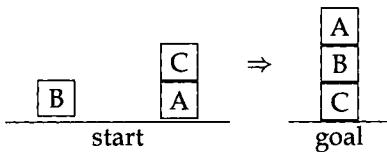
> (gps '((a on b) (b on c) (c on table) (space on a) (space on table))
         '((b on a) (c on b)))
((START)
 (EXECUTING (MOVE A FROM B TO TABLE))
 (EXECUTING (MOVE B FROM C TO A))
 (EXECUTING (MOVE C FROM TABLE TO B)))

> (gps '((a on b) (b on c) (c on table) (space on a) (space on table))
         '((c on b) (b on a)))
((START)
 (EXECUTING (MOVE A FROM B TO TABLE))
 (EXECUTING (MOVE B FROM C TO A))
 (EXECUTING (MOVE C FROM TABLE TO B)))

```

### The Sussman Anomaly

Surprisingly, there are problems that can't be solved by *any* reordering of goals. Consider:



This doesn't look too hard, so let's see how our GPS handles it:

```
> (setf start '((c on a) (a on table) (b on table) (space on c)
                (space on b) (space on table)))
((C ON A) (A ON TABLE) (B ON TABLE) (SPACE ON C)
 (SPACE ON B) (SPACE ON TABLE))

> (gps start '((a on b) (b on c))) => NIL
> (gps start '((b on c) (a on b))) => NIL
```

There is a “prerequisite clobbers sibling goal” problem regardless of which way we order the conjuncts! In other words, no combination of plans for the two individual goals can solve the conjunction of the two goals. This is a surprising fact, and the example has come to be known as “the Sussman anomaly.”<sup>4</sup> We will return to this problem in chapter 6.

## 4.15 Stage 5 Repeated: Analysis of Version 2

We have shown that GPS is extensible to multiple domains. The main point is that we didn't need to change the program itself to get the new domains to work; we just changed the list of operators passed to GPS. Experience in different domains did suggest changes that could be made, and we showed how to incorporate a few changes. Although version 2 is a big improvement over version 1, it still leaves much to be desired. Now we will discover a few of the most troubling problems.

---

<sup>4</sup>A footnote in Waldinger 1977 says, “This problem was proposed by Allen Brown. Perhaps many children thought of it earlier but did not recognize that it was hard.” The problem is named after Gerald Sussman because he popularized it in Sussman 1973.

## 4.16 The Not Looking after You Don't Leap Problem

We solved the “leaping before you look” problem by introducing variables to hold a representation of possible future states, rather than just a single variable representing the current state. This prevents GPS from taking an ill-advised action, but we shall see that even with all the repair strategies introduced in the last section, it doesn’t guarantee that a solution will be found whenever one is possible.

To see the problem, add another operator to the front of the \*school-ops\* list and turn the debugging output back on:

```
(use (push (op 'taxi-son-to-school
               :preconds '(son-at-home have-money)
               :add-list '(son-at-school)
               :del-list '(son-at-home have-money))
           *school-ops*))

(debug :gps)
```

Now, consider the problem of getting the child to school without using any money:

```
> (gps '(son-at-home have-money car-works)
      '(son-at-school have-money))
Goal: SON-AT-SCHOOL
Consider: TAXI-SON-TO-SCHOOL
Goal: SON-AT-HOME
Goal: HAVE-MONEY
Action: TAXI-SON-TO-SCHOOL
Goal: HAVE-MONEY
Goal: HAVE-MONEY
Goal: SON-AT-SCHOOL
Consider: TAXI-SON-TO-SCHOOL
Goal: SON-AT-HOME
Goal: HAVE-MONEY
Action: TAXI-SON-TO-SCHOOL
NIL
```

The first five lines of output successfully solve the son-at-school goal with the TAXI-SON-TO-SCHOOL action. The next line shows an unsuccessful attempt to solve the have-money goal. The next step is to try the other ordering. This time, the have-money goal is tried first, and succeeds. Then, the son-at-school goal is achieved again by the TAXI-SON-TO-SCHOOL action. But the check for consistency in achieve-each fails, and there are no repairs available. The goal fails, even though there is a valid solution: driving to school.

The problem is that `achieve` uses `some` to look at the appropriate-ops. Thus, if there is some appropriate operator, `achieve` succeeds. If there is only one goal, this will yield a correct solution. However, if there are multiple goals, as in this case, `achieve` will still only find one way to fulfill the first goal. If the first solution is a bad one, the only recourse is to try to repair it. In domains like the block world and maze world, repair often works, because all steps are reversible. But in the taxi example, no amount of plan repair can get the money back once it is spent, so the whole plan fails.

There are two ways around this problem. The first approach is to examine all possible solutions, not just the first solution that achieves each subgoal. The language Prolog, to be discussed in chapter 11, does just that. The second approach is to have `achieve` and `achieve-all` keep track of a list of goals that must be *protected*. In the taxi example, we would trivially achieve the `have-money` goal and then try to achieve `son-at-school`, while protecting the goal `have-money`. An operator would only be appropriate if it didn't delete any protected goals. This approach still requires some kind of repair or search through multiple solution paths. If we tried only one ordering—achieving `son-at-school` and then trying to protect it while achieving `have-money`—then we would not find the solution. David Warren's WARPLAN planner makes good use of the idea of protected goals.

## 4.17 The Lack of Descriptive Power Problem

It would be a lot more economical, in the maze domain, to have one operator that says we can move from here to there if we are at "here," and if there is a connection from "here" to "there." Then the input to a particular problem could list the valid connections, and we could solve any maze with this single operator. Similarly, we have defined an operator where the monkey pushes the chair from the door to the middle of the room, but it would be better to have an operator where the monkey can push the chair from wherever it is to any other nearby location, or better yet, an operator to push any "pushable" object from one location to a nearby one, as long as there is no intervening obstacle. The conclusion is that we would like to have variables in the operators, so we could say something like:

```
(op '(push X from A to B)
    :preconds '((monkey at A) (X at A) (pushable X) (path A B))
    :add-list '((monkey at B) (X at B))
    :del-list '((monkey at A) (X at A)))
```

Often we want to characterize a state in terms of something more abstract than a list of conditions. For example, in solving a chess problem, the goal is to have the opponent in checkmate, a situation that cannot be economically described in terms of primitives like (`black king on A 4`), so we need to be able to state some kind

of constraint on the goal state, rather than just listing its components. We might want to be able to achieve a disjunction or negation of conditions, where the current formalism allows only a conjunction.

It also is important, in many domains, to be able to state problems dealing with time: we want to achieve X before time  $T_0$ , and then achieve Y before time  $T_2$ , but not before  $T_1$ . Scheduling work on a factory floor or building a house are examples of planning where time plays an important role.

Often there are costs associated with actions, and we want to find a solution with minimal, or near-minimal costs. The cost might be as simple as the number of operators required for a solution—we saw in the blocks world domain that sometimes an operator that could be applied immediately was ignored, and an operator that needed several preconditions satisfied was chosen instead. Or we may be satisfied with a partial solution, if a complete solution is impossible or too expensive. We may also want to take the cost (and time) of computation into account.

## 4.18 The Perfect Information Problem

All the operators we have seen so far have unambiguous results; they add or delete certain things from the current state, and GPS always knows exactly what they are going to do. In the real world, things are rarely so cut and dried. Going back to the problem of becoming rich, one relevant operator would be playing the lottery. This operator has the effect of consuming a few dollars, and once in a while paying off a large sum. But we have no way to represent a payoff “once in a while.” Similarly, we have no way to represent unexpected difficulties of any kind. In the nursery school problem, we could represent the problem with the car battery by having GPS explicitly check to see if the car was working, or if it needed a battery, every time the program considered the driving operator. In the real world, we are seldom this careful; we get in the car, and only when it doesn’t start do we consider the possibility of a dead battery.

## 4.19 The Interacting Goals Problem

People tend to have multiple goals, rather than working on one at a time. Not only do I want to get the kid to nursery school, but I want to avoid getting hit by another car, get to my job on time, get my work done, meet my friends, have some fun, continue breathing, and so on. I also have to discover goals on my own, rather than work on a set of predefined goals passed to me by someone else. Some goals I can keep in the background for years, and then work on them when the opportunity presents itself. There is never a notion of satisfying all possible goals. Rather, there is a

continual process of achieving some goals, partially achieving others, and deferring or abandoning still others.

In addition to having active goals, people also are aware of undesirable situations that they are trying to avoid. For example, suppose I have a goal of visiting a friend in the hospital. This requires being at the hospital. One applicable operator might be to walk to the hospital, while another would be to severely injure myself and wait for the ambulance to take me there. The second operator achieves the goal just as well (perhaps faster), but it has an undesirable side effect. This could be addressed either with a notion of solution cost, as outlined in the last section, or with a list of background goals that every solution attempts to protect.

Herb Simon coined the term “satisficing” to describe the strategy of satisfying a reasonable number of goals to a reasonable degree, while abandoning or postponing other goals. GPS only knows success and failure, and thus has no way of maximizing partial success.

## 4.20 The End of GPS

These last four sections give a hint as to the scope of the limitations of GPS. In fact, it is not a very general problem solver at all. It is general in the sense that the algorithm is not tied to a particular domain; we can change domain by changing the operators. But GPS fails to be general in that it can't solve many interesting problems. It is confined to small tricks and games.

There is an important yet subtle reason why GPS was destined to fail, a reason that was not widely appreciated in 1957 but now is at the core of computer science. It is now recognized that there are problems that computers can't solve—not because a theoretically correct program can't be written, but because the execution of the program will take too long. A large number of problems can be shown to fall into the class of “NP-hard” problems. Computing a solution to these problems takes time that grows exponentially as the size of the problem grows. This is a property of the problems themselves, and holds no matter how clever the programmer is. Exponential growth means that problems that can be solved in seconds for, say, a five-input case may take trillions of years when there are 100 inputs. Buying a faster computer won't help much. After all, if a problem would take a trillion years to solve on your computer, it won't help much to buy 1000 computers each 1000 times faster than the one you have: you're still left with a million years wait. For a theoretical computer scientist, discovering that a problem is NP-hard is an end in itself. But for an AI worker, it means that the wrong question is being asked. Many problems are NP-hard when we insist on the optimal solution but are much easier when we accept a solution that might not be the best.

The input to GPS is essentially a program, and the execution of GPS is the execution of that program. If GPS's input language is general enough to express any program,

then there will be problems that can't be solved, either because they take too long to execute or because they have no solution. Modern problem-solving programs recognize this fundamental limitation, and either limit the class of problems they try to solve or consider ways of finding approximate or partial solutions. Some problem solvers also monitor their own execution time and know enough to give up when a problem is too hard.

The following quote from Drew McDermott's article "Artificial Intelligence Meets Natural Stupidity" sums up the current feeling about GPS. Keep it in mind the next time you have to name a program.

*Remember GPS? By now, "GPS" is a colorless term denoting a particularly stupid program to solve puzzles. But it originally meant "General Problem Solver," which caused everybody a lot of needless excitement and distraction. It should have been called LFGNS—"Local Feature-Guided Network Searcher."*

Nonetheless, GPS has been a useful vehicle for exploring programming in general, and AI programming in particular. More importantly, it has been a useful vehicle for exploring "the nature of deliberation." Surely we'll admit that Aristotle was a smarter person than you or me, yet with the aid of the computational model of mind as a guiding metaphor, and the further aid of a working computer program to help explore the metaphor, we have been led to a more thorough appreciation of means-ends analysis—at least within the computational model. We must resist the temptation to believe that all thinking follows this model.

The appeal of AI can be seen as a split between means and ends. The end of a successful AI project can be a program that accomplishes some useful task better, faster, or cheaper than it could be before. By that measure, GPS is a mostly a failure, as it doesn't solve many problems particularly well. But the means toward that end involved an investigation and formalization of the problem-solving process. By that measure, our reconstruction of GPS is a success to the degree in which it leads the reader to a better understanding of the issues.

## 4.21 History and References

The original GPS is documented in Newell and Simon's 1963 paper and in their 1972 book, *Human Problem Solving*, as well as in Ernst and Newell 1969. The implementation in this chapter is based on the STRIPS program (Fikes and Nilsson 1971).

There are other important planning programs. Earl Sacerdoti's ABSTRIPS program was a modification of STRIPS that allowed for hierarchical planning. The idea was to sketch out a skeletal plan that solved the entire program at an abstract level, and then fill in the details. David Warren's WARPLAN planner is covered in Warren 1974a,b and in a section of Coelho and Cotta 1988. Austin Tate's NONLIN system (Tate 1977)

achieved greater efficiency by considering a plan as a partially ordered sequence of operations rather than as a strictly ordered sequence of situations. David Chapman's TWEAK synthesizes and formalizes the state of the art in planning as of 1987.

All of these papers—and quite a few other important planning papers—are reprinted in Allen, Hendler, and Tate 1990.

## 4.22 Exercises

- ❑ **Exercise 4.1 [m]** It is possible to implement dbg using a single call to format. Can you figure out the format directives to do this?
- ❑ **Exercise 4.2 [m]** Write a function that generates all permutations of its input.
- ❑ **Exercise 4.3 [h]** GPS does not recognize the situation where a goal is accidentally solved as part of achieving another goal. Consider the goal of eating dessert. Assume that there are two operators available: eating ice cream (which requires having the ice cream) and eating cake (which requires having the cake). Assume that we can buy a cake, and that the bakery has a deal where it gives out free ice cream to each customer who purchases and eats a cake. (1) Design a list of operators to represent this situation. (2) Give gps the goal of eating dessert. Show that, with the right list of operators, gps will decide to eat ice cream, then decide to buy and eat the cake in order to get the free ice cream, and then go ahead and eat the ice cream, even though the goal of eating dessert has already been achieved by eating the cake. (3) Fix gps so that it does not manifest this problem.

The following exercises address the problems in version 2 of the program.

- ❑ **Exercise 4.4 [h]** *The Not Looking after You Don't Leap Problem.* Write a program that keeps track of the remaining goals so that it does not get stuck considering only one possible operation when others will eventually lead to the goal. Hint: have achieve take an extra argument indicating the goals that remain to be achieved after the current goal is achieved. achieve should succeed only if it can achieve the current goal and also achieve-all the remaining goals.
- ❑ **Exercise 4.5 [d]** Write a planning program that, like Warren's WARPLAN, keeps track of the list of goals that remain to be done as well as the list of goals that have been achieved and should not be undone. The program should never undo a goal that has been achieved, but it should allow for the possibility of reordering steps that

have already been taken. In this way, the program will solve the Sussman anomaly and similar problems.

- ?
- Exercise 4.6 [d]** *The Lack of Descriptive Power Problem.* Read chapters 5 and 6 to learn about pattern matching. Write a version of GPS that uses the pattern matching tools, and thus allows variables in the operators. Apply it to the maze and blocks world domains. Your program will be more efficient if, like Chapman's TWEAK program, you allow for the possibility of variables that remain unbound as long as possible.
  
- ?
- Exercise 4.7 [d]** Speculate on the design of a planner that can address the *Perfect Information* and *Interacting Goals* problems.

## 4.23 Answers

**Answer 4.1** In this version, the format string "`~&~V@T~?~`" breaks down as follows: "`~&`" means go to a fresh line; "`~V@T`" means insert spaces (`@T`) but use the next argument (`V`) to get the number of spaces. The "`~?`" is the indirection operator: use the next argument as a format string, and the argument following that as the list of arguments for the format string.

```
(defun dbg-indent (id indent format-string &rest args)
  "Print indented debugging info if (DEBUG ID) has been specified."
  (when (member id *dbg-ids*)
    (format *debug-io* "~&~V@T~?~" (* 2 indent) format-string args)))
```

**Answer 4.2** Here is one solution. The sophisticated Lisp programmer should also see the exercise on page 680.

```
(defun permutations (bag)
  "Return a list of all the permutations of the input."
  ;; If the input is nil, there is only one permutation:
  ;; nil itself
  (if (null bag)
      '())
  ;; Otherwise, take an element, e, out of the bag.
  ;; Generate all permutations of the remaining elements,
  ;; And add e to the front of each of these.
  ;; Do this for all possible e to generate all permutations.
  (mapcan #'(lambda (e)
              (mapcar #'(lambda (p) (cons e p))
                      (permutations
                        (remove e bag :count 1 :test #'eq))))
           bag)))
```

## CHAPTER 5

---

# ELIZA: Dialog with a Machine

*It is said that to explain is to explain away.*

—Joseph Weizenbaum  
MIT computer scientist

This chapter and the rest of part I will examine three more well-known AI programs of the 1960s. ELIZA held a conversation with the user in which it simulated a psychotherapist. STUDENT solved word problems of the kind found in high school algebra books, and MACSYMA solved a variety of symbolic mathematical problems, including differential and integral calculus. We will develop versions of the first two programs that duplicate most of the essential features, but for the third we will implement only a tiny fraction of the original program's capabilities.

All three programs make heavy use of a technique called pattern matching. Part I serves to show the versatility—and also the limitations—of this technique.

Of the three programs, the first two process input in plain English, and the last two solve non-trivial problems in mathematics, so there is some basis for describing them as being "intelligent." On the other hand, we shall see that this intelligence is largely an illusion, and that ELIZA in particular was actually designed to demonstrate this illusion, not to be a "serious" AI program.

ELIZA was one of the first programs to feature English output as well as input. The program was named after the heroine of *Pygmalion*, who was taught to speak proper English by a dedicated teacher. ELIZA's principal developer, MIT professor Joseph Weizenbaum, published a paper on ELIZA in the January 1966 issue of the *Communications of the Association for Computing Machinery*. The introduction to that paper is reproduced in its entirety here:

*It is said that to explain is to explain away. This maxim is nowhere so well fulfilled as in the area of computer programming, especially in what is called heuristic programming and artificial intelligence. For in those realms machines are made to behave in wondrous ways, often sufficient to dazzle even the most experienced observer. But once a particular program is unmasked, once its inner workings are explained in language sufficiently plain to induce understanding, its magic crumbles away; it stands revealed as a mere collection of procedures, each quite comprehensible. The observer says to himself, "I could have written that." With that thought he moves the program in question from the shelf marked "intelligent," to that reserved for curios, fit to be discussed only with people less enlightened than he.*

*The object of this paper is to cause just such a re-evaluation of the program about to be "explained." Few programs ever needed it more.*

Let me point out in passing that the main object of this book is to cause the reader to say to him or herself, "I could have written that." It is hoped that in some cases this will be because the reader has gained something, and not solely because the programs in question have lost their ability to dazzle.

Let's start our investigation of ELIZA by looking at a sample dialog from the Weizenbaum article. ELIZA was designed to be instructed by a variety of scripts, or patterns of interaction. In fact it had several scripts, including ones in German and Welsh. But the only script that was used to any extent was in English, and it made ELIZA emulate a Rogerian psychoanalyst. Rogerians are "nondirective"—they try to let the patient reveal himself. This is an easy script for ELIZA to follow, because it involves being passive and reacting, rather than volunteering new information. In the listing that follows, the user's input is in mixed case, and ELIZA's response is in capitals:

```
Men are all alike.  
IN WHAT WAY  
They're always bugging us about something or another.  
CAN YOU BE MORE SPECIFIC  
Well, my boyfriend made me come here.  
YOUR BOYFRIEND MADE YOU COME HERE  
He says I'm depressed much of the time.  
I AM SORRY TO HEAR YOU ARE DEPRESSED
```

It's true. I am unhappy.  
DO YOU THINK COMING HERE WILL HELP YOU NOT TO BE UNHAPPY  
I need some help, that much seems certain.  
WHAT WOULD IT MEAN TO YOU IF YOU GOT SOME HELP  
Perhaps I could learn to get along with my mother.  
TELL ME MORE ABOUT YOUR FAMILY

This seems quite impressive; the program is responding reasonably to a variety of inputs and appears to be exhibiting true "understanding." But, as Weizenbaum explains, this appearance is an illusion, and the program has finessed the difficult task of understanding by carefully recognizing, transforming, and echoing pieces of the input. The procedure is to look for specific patterns, based on a key word or words in the input. For example, whenever the program sees the words "alike" or "same," it prints a message asking about similarity, in this case, "IN WHAT WAY." Other patterns are more complicated. In the next to last exchange, ELIZA first throws away everything after the comma, and then uses a pattern that looks for input of the form "I need X" and transforms it to the output "What would it mean to you if you got X," where X is the phrase "some help" in this case. The great advantage of this approach is that the response will make sense for a variety of inputs: "I need a vacation" will elicit "What would it mean to you if you got a vacation." This will work automatically, without having to tell the program the difference between "some help" and "a vacation." The disadvantage is that the program will sometimes make silly responses: "I need this like a hole in the head" will be answered with "What would it mean to you if you got this like a hole in the head." In fact, the program only makes any sense at all if it is receiving input from someone who makes sense—and someone who interprets its answers charitably. When given "I need gibble plix zzq," ELIZA would reply "What would it mean to you if you got gibble plix zzq," without any inkling that something is wrong. Another example of the original ELIZA program has it communicating with another program, PARRY, which was written by Dr. Kenneth Colby to mimic paranoid behavior. Part of the conversation follows, with ELIZA again in capitals and PARRY in mixed case:

I don't understand your motives.  
WHY DON'T YOU UNDERSTAND MY MOTIVES?  
Shouldn't I be?  
WHY DO YOU SAY THAT JUST NOW?  
That's just the way it is.  
SUPPOSE THAT'S JUST THE WAY IT WERE NOT.  
So?  
I AM NOT SURE I UNDERSTAND YOU FULLY.  
You harp on the same questions too much.  
WHAT DOES THAT SIMILARITY SUGGEST TO YOU?  
I don't get you.  
DO YOU WISH THAT YOU DID GET I?

This suggests that neither ELIZA nor PARRY had a deep understanding of the dialog. When there is a human to take on most of the burden of carrying forth the conversation, these programs can be convincing, but when the human is taken out of the conversation, the weaknesses of the programs are revealed.

## 5.1 Describing and Specifying ELIZA

Now that we have an idea of what ELIZA is like, we can begin the description and specification of the program, and eventually move to the implementation and debugging.

The ELIZA algorithm can be described simply as: (1) read an input, (2) find a pattern that matches the input, (3) transform the input into a response, and (4) print the response. These four steps are repeated for each input.

The specification and implementation of steps (1) and (4) are trivial: for (1), use the built-in `read` function to read a list of words, and for (4) use `print` to print the list of words in the response.

Of course, there are some drawbacks to this specification. The user will have to type a real list—using parentheses—and the user can't use characters that are special to `read`, like quotation marks, commas, and periods. So our input won't be as unconstrained as in the sample dialog, but that's a small price to pay for the convenience of having half of the problem neatly solved.

## 5.2 Pattern Matching

The hard part comes with steps (2) and (3)—this notion of pattern matching and transformation. There are four things to be concerned with: a general pattern and response, and a specific input and transformation of that input. Since we have agreed to represent the input as a list, it makes sense for the other components to be lists too. For example, we might have:

Pattern: (i need a X)

Response: (what would it mean to you if you got a X ?)

Input: (i need a vacation)

Transformation: (what would it mean to you if you got a vacation ?)

The pattern matcher must match the literals `i` with `i`, `need` with `need`, and `a` with `a`, as well as match the variable `X` with `vacation`. This presupposes that there is some way of deciding that `X` is a variable and that `need` is not. We must then arrange to substitute `vacation` for `X` within the response, in order to get the final transformation.

Ignoring for a moment the problem of transforming the pattern into the response, we can see that this notion of pattern matching is just a generalization of the Lisp function `equal`. Below we show the function `simple-equal`, which is like the built-in function `equal`,<sup>1</sup> and the function `pat-match`, which is extended to handle pattern-matching variables:

```
(defun simple-equal (x y)
  "Are x and y equal? (Don't check inside strings.)"
  (if (or (atom x) (atom y))
      (eql x y)
      (and (simple-equal (first x) (first y))
            (simple-equal (rest x) (rest y)))))

(defun pat-match (pattern input)
  "Does pattern match input? Any variable can match anything."
  (if (variable-p pattern)
      t
      (if (or (atom pattern) (atom input))
          (eql pattern input)
          (and (pat-match (first pattern) (first input))
                (pat-match (rest pattern) (rest input))))))
```



**Exercise 5.1 [s]** Would it be a good idea to replace the complex and form in `pat-match` with the simpler (`(every #'pat-match pattern input)`)?

Before we can go on, we need to decide on an implementation for pattern-matching variables. We could, for instance, say that only a certain set of symbols, such as {X,Y,Z}, are variables. Alternately, we could define a structure of type `variable`, but then we'd have to type something verbose like `(make-variable :name 'X)` every time we wanted one. Another choice would be to use symbols, but to distinguish variables from constants by the name of the symbol. For example, in Prolog, variables start with capital letters and constants with lowercase. But Common Lisp is case-insensitive, so that won't work. Instead, there is a tradition in Lisp-based AI programs to have variables be symbols that start with the question mark character.

So far we have dealt with symbols as atoms—objects with no internal structure. But things are always more complicated than they first appear and, as in Lisp as in physics, it turns out that even atoms have components. In particular, symbols have names, which are strings and are accessible through the `symbol-name` function. Strings in turn have elements that are characters, accessible through the function `char`. The character '?' is denoted by the self-evaluating escape sequence `#\?`. So the predicate `variable-p` can be defined as follows, and we now have a complete pattern matcher:

---

<sup>1</sup>The difference is that `simple-equal` does not handle strings.

```
(defun variable-p (x)
  "Is x a variable (a symbol beginning with '?')?"
  (and (symbolp x) (equal (char (symbol-name x) 0) #\?)))

> (pat-match '(I need a ?X) '(I need a vacation))
T

> (pat-match '(I need a ?X) '(I really need a vacation))
NIL
```

In each case we get the right answer, but we don't get any indication of what ?X is, so we couldn't substitute it into the response. We need to modify pat-match to return some kind of table of variables and corresponding values. In making this choice, the experienced Common Lisp programmer can save some time by being opportunistic: recognizing when there is an existing function that will do a large part of the task at hand. What we want is to substitute values for variables throughout the response. The alert programmer could refer to the index of this book or the Common Lisp reference manual and find the functions substitute, subst, and sublis. All of these substitute some new expression for an old one within an expression. It turns out that sublis is most appropriate because it is the only one that allows us to make several substitutions all at once. sublis takes two arguments, the first a list of old-new pairs, and the second an expression in which to make the substitutions. For each one of the pairs, the car is replaced by the cdr. In other words, we would form each pair with something like (cons old new). (Such a list of pairs is known as an *association list*, or *a-list*, because it associates keys with values. See section 3.6.) In terms of the example above, we would use:

```
> (sublis '((?X . vacation))
           '(what would it mean to you if you got a ?X ?))
(WHAT WOULD IT MEAN TO YOU IF YOU GOT A VACATION ?)
```

Now we need to arrange for pat-match to return an a-list, rather than just T for success. Here's a first attempt:

```
(defun pat-match (pattern input)
  "Does pattern match input? WARNING: buggy version."
  (if (variable-p pattern)
      (list (cons pattern input))
      (if (or (atom pattern) (atom input))
          (eql pattern input)
          (append (pat-match (first pattern) (first input))
                  (pat-match (rest pattern) (rest input))))))
```

This implementation looks reasonable: it returns an a-list of one element if the pattern is a variable, and it appends alists if the pattern and input are both lists. However,

there are several problems. First, the test (`(eq pattern input)`) may return `T`, which is not a list, so `append` will complain. Second, the same test might return `nil`, which should indicate failure, but it will just be treated as a list, and will be appended to the rest of the answer. Third, we haven't distinguished between the case where the match fails—and returns `nil`—versus the case where everything matches, but there are no variables, so it returns the null a-list. (This is the semipredicate problem discussed on page 127.) Fourth, we want the bindings of variables to agree—if `?X` is used twice in the pattern, we don't want it to match two different values in the input. Finally, it is inefficient for `pat-match` to check both the `first` and `rest` of lists, even when the corresponding `first` parts fail to match. (Isn't it amazing that there could be five bugs in a seven-line function?)

We can resolve these problems by agreeing on two major conventions. First, it is very convenient to make `pat-match` a true predicate, so we will agree that it returns `nil` only to indicate failure. That means that we will need a non-`nil` value to represent the empty binding list. Second, if we are going to be consistent about the values of variables, then the `first` will have to know what the `rest` is doing. We can accomplish this by passing the binding list as a third argument to `pat-match`. We make it an optional argument, because we want to be able to say simply `(pat-match a b)`.

To abstract away from these implementation decisions, we define the constants `fail` and `no-bindings` to represent the two problematic return values. The special form `defconstant` is used to indicate that these values will not change. (It is customary to give special variables names beginning and ending with asterisks, but this convention usually is not followed for constants. The reasoning is that asterisks shout out, "Careful! I may be changed by something outside of this lexical scope." Constants, of course, will not be changed.)

```
(defconstant fail nil "Indicates pat-match failure")
(defconstant no-bindings '((t . t))
  "Indicates pat-match success, with no variables.")
```

Next, we abstract away from `assoc` by introducing the following four functions:

```
(defun get-binding (var bindings)
  "Find a (variable . value) pair in a binding list."
  (assoc var bindings))

(defun binding-val (binding)
  "Get the value part of a single binding."
  (cdr binding))

(defun lookup (var bindings)
  "Get the value part (for var) from a binding list."
  (binding-val (get-binding var bindings)))
```

```
(defun extend-bindings (var val bindings)
  "Add a (var . value) pair to a binding list."
  (cons (cons var val) bindings))
```

Now that variables and bindings are defined, pat-match is easy. It consists of five cases. First, if the binding list is fail, then the match fails (because some previous match must have failed). If the pattern is a single variable, then the match returns whatever match-variable returns; either the existing binding list, an extended one, or fail. Next, if both pattern and input are lists, we first call pat-match recursively on the first element of each list. This returns a binding list (or fail), which we use to match the rest of the lists. This is the only case that invokes a nontrivial function, so it is a good idea to informally prove that the function will terminate: each of the two recursive calls reduces the size of both pattern and input, and pat-match checks the case of atomic patterns and inputs, so the function as a whole must eventually return an answer (unless both pattern and input are of infinite size). If none of these four cases succeeds, then the match fails.

```
(defun pat-match (pattern input &optional (bindings no-bindings))
  "Match pattern against input in the context of the bindings"
  (cond ((eq bindings fail) fail)
        ((variable-p pattern)
         (match-variable pattern input bindings))
        ((eql pattern input) bindings)
        ((and (consp pattern) (consp input))
         (pat-match (rest pattern) (rest input)
                    (pat-match (first pattern) (first input)
                               bindings)))
        (t fail)))

(defun match-variable (var input bindings)
  "Does VAR match input? Uses (or updates) and returns bindings."
  (let ((binding (get-binding var bindings)))
    (cond ((not binding) (extend-bindings var input bindings))
          ((equal input (binding-val binding)) bindings)
          (t fail))))
```

We can now test pat-match and see how it works:

```
> (pat-match '(i need a ?X) '(i need a vacation))
((?X . VACATION) (T . T))
```

The answer is a list of variable bindings in dotted pair notation; each element of the list is a (*variable* . *value*) pair. The (T . T) is a remnant from no-bindings. It does no real harm, but we can eliminate it by making extend-bindings a little more complicated:

```
(defun extend-bindings (var val bindings)
  "Add a (var . value) pair to a binding list."
  (cons (cons var val)
        (if (eq bindings no-bindings)
            nil
            bindings))

> (sublis (pat-match '(i need a ?X) '(i need a vacation))
           '(what would it mean to you if you got a ?X ?))
(WHAT WOULD IT MEAN TO YOU IF YOU GOT A VACATION ?)

> (pat-match '(i need a ?X) '(i really need a vacation))
NIL

> (pat-match '(this is easy) '(this is easy))
((T . T))

> (pat-match '(%X is ?X) '((2 + 2) is 4))
NIL

> (pat-match '(%X is ?X) '((2 + 2) is (2 + 2)))
((?X 2 + 2))

> (pat-match '(%P need . ?X) '(i need a long vacation))
((?X A LONG VACATION) (?P . I))
```

Notice the distinction between NIL and ((T . T)). The latter means that the match succeeded, but there were no bindings to return. Also, remember that (?X 2 + 2) means the same as (?X . (2 + 2)).

A more powerful implementation of pat-match is given in chapter 6. Yet another implementation is given in section 10.4. It is more efficient but more cumbersome to use.

## 5.3 Segment Pattern Matching

In the pattern (?P need . ?X), the variable ?X matches the rest of the input list, regardless of its length. This is in contrast to ?P, which can only match a single element, namely, the first element of the input. For many applications of pattern matching, this is fine; we only want to match corresponding elements. However, ELIZA is somewhat different in that we need to account for variables in any position that match a sequence of items in the input. We will call such variables *segment variables*. We will need a notation to differentiate segment variables from normal

variables. The possibilities fall into two classes: either we use atoms to represent segment variables and distinguish them by some spelling convention (as we did to distinguish variables from constants) or we use a nonatomic construct. We will choose the latter, using a list of the form `(?* variable)` to denote segment variables. The symbol `?*` is chosen because it combines the notion of variable with the Kleene-star notation. So, the behavior we want from pat-match is now:

```
> (pat-match '((?* ?p) need (?* ?x))
  '(Mr Hulot and I need a vacation))
((?P MR HULOT AND I) (?X A VACATION))
```

In other words, when both pattern and input are lists and the first element of the pattern is a segment variable, then the variable will match some initial part of the input, and the rest of the pattern will attempt to match the rest. We can update pat-match to account for this by adding a single cond-clause. Defining the predicate to test for segment variables is also easy:

```
(defun pat-match (pattern input &optional (bindings no-bindings))
  "Match pattern against input in the context of the bindings"
  (cond ((eq bindings fail) fail)
        ((variable-p pattern)
         (match-variable pattern input bindings))
        ((eql pattern input) bindings)
        ((segment-pattern-p pattern) ; ***
         (segment-match pattern input bindings)) ; ***
        ((and (consp pattern) (consp input))
         (pat-match (rest pattern) (rest input)
                    (pat-match (first pattern) (first input)
                               bindings)))
        (t fail)))
(defun segment-pattern-p (pattern)
  "Is this a segment matching pattern: ((?* var) . pat)"
  (and (consp pattern)
       (starts-with (first pattern) '?*)))
```

In writing segment-match, the important question is how much of the input the segment variable should match. One answer is to look at the next element of the pattern (the one after the segment variable) and see at what position it occurs in the input. If it doesn't occur, the total pattern can never match, and we should fail. If it does occur, call its position pos. We will want to match the variable against the initial part of the input, up to pos. But first we have to see if the rest of the pattern matches the rest of the input. This is done by a recursive call to pat-match. Let the result of this recursive call be named b2. If b2 succeeds, then we go ahead and match the segment variable against the initial subsequence.

The tricky part is when `b2` fails. We don't want to give up completely, because it may be that if the segment variable matched a longer subsequence of the input, then the rest of the pattern would match the rest of the input. So what we want is to try `segment-match` again, but forcing it to consider a longer match for the variable. This is done by introducing an optional parameter, `start`, which is initially 0 and is increased with each failure. Notice that this policy rules out the possibility of any kind of variable following a segment variable. (Later we will remove this constraint.)

```
(defun segment-match (pattern input bindings &optional (start 0))
  "Match the segment pattern ((?* var) . pat) against input."
  (let ((var (second (first pattern)))
        (pat (rest pattern)))
    (if (null pat)
        (match-variable var input bindings)
        ;; We assume that pat starts with a constant
        ;; In other words, a pattern can't have 2 consecutive vars
        (let ((pos (position (first pat) input
                             :start start :test #'equal)))
          (if (null pos)
              fail
              (let ((b2 (pat-match pat (subseq input pos) bindings)))
                ;; If this match failed, try another longer one
                ;; If it worked, check that the variables match
                (if (eq b2 fail)
                    (segment-match pattern input bindings (+ pos 1))
                    (match-variable var (subseq input 0 pos) b2)))))))
```

Some examples of segment matching follow:

```
> (pat-match '((?* ?p) need (?* ?x))
             '(Mr Hulot and I need a vacation))
((?P MR HULOT AND I) (?X A VACATION))

> (pat-match '((?* ?x) is a (?* ?y)) '(what he is is a fool))
((?X WHAT HE IS) (?Y FOOL))
```

The first of these examples shows a fairly simple case: `?p` matches everything up to `need`, and `?x` matches the rest. The next example involves the more complicated backup case. First `?x` matches everything up to the first `is` (this is position 2, since counting starts at 0 in Common Lisp). But then the pattern `a` fails to match the input `is`, so `segment-match` tries again with starting position 3. This time everything works; `is` matches `is`, `a` matches `a`, and `(?* ?y)` matches `fool`.

Unfortunately, this version of `segment-match` does not match as much as it should. Consider the following example:

```
> (pat-match '((?* ?x) a b (?* ?x)) '(1 2 a b a b 1 2 a b)) => NIL
```

This fails because `?x` is matched against the subsequence `(1 2)`, and then the remaining pattern successfully matches the remaining input, but the final call to `match-variable` fails, because `?x` has two different values. The fix is to call `match-variable` before testing whether the `b2` fails, so that we will be sure to try `segment-match` again with a longer match no matter what the cause of the failure.

```
(defun segment-match (pattern input bindings &optional (start 0))
  "Match the segment pattern ((?* var) . pat) against input."
  (let ((var (second (first pattern)))
        (pat (rest pattern)))
    (if (null pat)
        (match-variable var input bindings)
        ;; We assume that pat starts with a constant
        ;; In other words, a pattern can't have 2 consecutive vars
        (let ((pos (position (first pat) input
                             :start start :test #'equal)))
          (if (null pos)
              fail
              (let ((b2 (pat-match
                         pat (subseq input pos)
                         (match-variable var (subseq input 0 pos)
                                         bindings))))
                  ;; If this match failed, try another longer one
                  (if (eq b2 fail)
                      (segment-match pattern input bindings (+ pos 1))
                      b2)))))))
```

Now we see that the match goes through:

```
> (pat-match '((?* ?x) a b (?* ?x)) '(1 2 a b a b 1 2 a b))
((?X 1 2 A B))
```

Note that this version of `segment-match` tries the shortest possible match first. It would also be possible to try the longest match first.

## 5.4 The ELIZA Program: A Rule-Based Translator

Now that we have a working pattern matcher, we need some patterns to match. What's more, we want the patterns to be associated with responses. We can do this by inventing a data structure called a rule, which consists of a pattern and one or more associated responses. These are rules in the sense that they assert, "If you see A, then respond with B or C, chosen at random." We will choose the simplest possible implementation for rules: as lists, where the first element is the pattern and the rest is a list of responses:

```
(defun rule-pattern (rule) (first rule))
(defun rule-responses (rule) (rest rule))
```

Here's an example of a rule:

```
(((* ?x) I want (* ?y))
 (What would it mean if you got ?y)
 (Why do you want ?y)
 (Suppose you got ?y soon))
```

When applied to the input (*I want to test this program*), this rule (when interpreted by the ELIZA program) would pick a response at random, substitute in the value of *?y*, and respond with, say, (*why do you want to test this program*).

Now that we know what an individual rule will do, we need to decide how to handle a set of rules. If ELIZA is to be of any interest, it will have to have a variety of responses. So several rules may all be applicable to the same input. One possibility would be to choose a rule at random from among the rules having patterns that match the input.

Another possibility is just to accept the first rule that matches. This implies that the rules form an ordered list, rather than an unordered set. The clever ELIZA rule writer can take advantage of this ordering and arrange for the most specific rules to come first, while more vague rules are near the end of the list.

The original ELIZA had a system where each rule had a priority number associated with it. The matching rule with the highest priority was chosen. Note that putting the rules in order achieves the same effect as having a priority number on each rule: the first rule implicitly has the highest priority, the second rule is next highest, and so on.

Here is a short list of rules, selected from Weizenbaum's original article, but with the form of the rules updated to the form we are using. The answer to exercise 5.19 contains a longer list of rules.

```
(defparameter *eliza-rules*
'(((?* ?x) hello (?* ?y))
  (How do you do. Please state your problem.))
(((?* ?x) I want (?* ?y))
  (What would it mean if you got ?y)
  (Why do you want ?y) (Suppose you got ?y soon))
(((?* ?x) if (?* ?y))
  (Do you really think its likely that ?y) (Do you wish that ?y)
  (What do you think about ?y) (Really-- if ?y))
(((?* ?x) no (?* ?y))
  (Why not?) (You are being a bit negative)
  (Are you saying "NO" just to be negative?))
(((?* ?x) I was (?* ?y))
  (Were you really?) (Perhaps I already knew you were ?y)
  (Why do you tell me you were ?y now?))
(((?* ?x) I feel (?* ?y))
  (Do you often feel ?y ?))
(((?* ?x) I felt (?* ?y)))
  (What other feelings do you have?))))
```

Finally we are ready to define ELIZA proper. As we said earlier, the main program should be a loop that reads input, transforms it, and prints the result. Transformation is done primarily by finding some rule such that its pattern matches the input, and then substituting the variables into the rule's response. The program is summarized in figure 5.1.

There are a few minor complications. We print a prompt to tell the user to input something. We use the function `flatten` to insure that the output won't have imbedded lists after variable substitution. An important trick is to alter the input by swapping "you" for "me" and so on, since these terms are relative to the speaker. Here is the complete program:

```
(defun eliza ()
  "Respond to user input using pattern matching rules."
  (loop
    (print 'eliza>)
    (write (flatten (use-eliza-rules (read))) :pretty t)))

(defun use-eliza-rules (input)
  "Find some rule with which to transform the input."
  (some #'(lambda (rule)
            (let ((result (pat-match (rule-pattern rule) input)))
              (if (not (eq result fail))
                  (sublis (switch-viewpoint result)
                          (random-elt (rule-responses rule)))))))
        *eliza-rules*))
```

	<b>Top-Level Function</b>
eliza	Respond to user input using pattern matching rules.
	<b>Special Variables</b>
*eliza-rules*	A list of transformation rules.
	<b>Data Types</b>
rule	An association of a pattern with a list of responses.
	<b>Functions</b>
eliza	Respond to user input using pattern matching rules.
use-eliza-rules	Find some rule with which to transform the input.
switch-viewpoint	Change I to you and vice versa, and so on.
flatten	Append together elements of a list.
	<b>Selected Common Lisp Functions</b>
sublis	Substitute elements into a tree.
	<b>Previously Defined Functions</b>
random-elt	Pick a random element from a list. (p. 36)
pat-match	Match a pattern against an input. (p. 160)
mappend	Append together the results of a mapcar.

Figure 5.1: Glossary for the ELIZA Program

```
(defun switch-viewpoint (words)
  "Change I to you and vice versa, and so on."
  (sublis '((I . you) (you . I) (me . you) (am . are))
          words))
```

Note the use of `write` with the `:pretty` keyword `true`. This will give better formatted output in some cases. The program makes use of the previously defined `random-elt`, and `flatten`, which is defined here using `mappend` and `mklist`, a function that is defined in the InterLisp dialect but not in Common Lisp.

```
(defun flatten (the-list)
  "Append together elements (or lists) in the list."
  (mappend #'mklist the-list))

(defun mklist (x)
  "Return x if it is a list, otherwise (x)."
  (if (listp x)
      x
      (list x)))

(defun mappend (fn the-list)
  "Apply fn to each element of list and append the results."
  (apply #'append (mapcar fn the-list)))
```

```
(defun random-elt (choices)
  "Choose an element from a list at random."
  (elt choices (random (length choices))))
```

The actual ELIZA program had a few complications that don't show up in this version. First, there was an alias method for associating several words with the same pattern; both "mother" and "father" could be associated with the "family" pattern. There was also a synonym mechanism that would treat "don't" and "do not" or "everybody" and "everyone" the same. In the case of input with several comma-separated phrases, each phrase was processed separately, and the response with the highest priority was selected. In addition, the original ELIZA had a "memory" mechanism. When no pattern matched the input, it said something like "Tell me more about X," where X is some input given earlier. Finally, our version has fewer rules. Despite the shortcomings of our version, it can occasionally hold up its end of a conversation, as shown below:

```
> (eliza)
ELIZA> (hello there)
(HOW DO YOU DO. PLEASE STATE YOUR PROBLEM.)
ELIZA> (i want to test this program)
(WHAT WOULD IT MEAN IF YOU GOT TO TEST THIS PROGRAM)
ELIZA> (i could see if it works)
(DO YOU REALLY THINK ITS LIKELY THAT IT WORKS)
ELIZA> (no not really)
(ARE YOU SAYING "NO" JUST TO BE NEGATIVE?)
ELIZA> (no)
(ARE YOU SAYING "NO" JUST TO BE NEGATIVE?)
ELIZA> (forget it-- i was wondering how general the program is)
(WHY DO YOU TELL ME YOU WERE WONDERING HOW GENERAL THE PROGRAM IS NOW?)
ELIZA> (i felt like it)
(WHAT OTHER FEELINGS DO YOU HAVE?)
ELIZA> (i feel this is enough)
(DO YOU OFTEN FEEL THIS IS ENOUGH ?)
ELIZA> [Abort]
>
```

In the end, it is the technique that is important—not the program. ELIZA has been "explained away" and should rightfully be moved to the curio shelf. Pattern matching in general remains important technique, and we will see it again in subsequent chapters. The notion of a rule-based translator is also important. The problem of understanding English (and other languages) remains an important part of AI. Clearly, the problem of understanding English is not solved by ELIZA. In part V, we will address the problem again, using more sophisticated techniques.

## 5.5 History and References

As mentioned above, the original article describing ELIZA is Weizenbaum 1966. Another dialog system using similar pattern-matching techniques is Kenneth Colby's (1975) PARRY. This program simulated the conversation of a paranoid person well enough to fool several professional psychologists. Although the pattern matching techniques were simple, the model of belief maintained by the system was much more sophisticated than ELIZA. Colby has suggested that dialog programs like ELIZA, augmented with some sort of belief model like PARRY, could be useful tools in treating mentally disturbed people. According to Colby, it would be inexpensive and effective to have patients converse with a specially designed program, one that could handle simple cases and alert doctors to patients that needed more help. Weizenbaum's book *Computer Power and Human Reason* (1976) discusses ELIZA and PARRY and takes a very critical view toward Colby's suggestion. Other interesting early work on dialog systems that model belief is reported by Allan Collins (1978) and Jamie Carbonell (1981).

## 5.6 Exercises

-  **Exercise 5.2 [m]** Experiment with this version of ELIZA. Show some exchanges where it performs well, and some where it fails. Try to characterize the difference. Which failures could be fixed by changing the rule set, which by changing the pat-match function (and the pattern language it defines), and which require a change to the eliza program itself?
-  **Exercise 5.3 [h]** Define a new set of rules that make ELIZA give stereotypical responses to some situation other than the doctor–patient relationship. Or, write a set of rules in a language other than English. Test and debug your new rule set.
-  **Exercise 5.4 [s]** We mentioned that our version of ELIZA cannot handle commas or double quote marks in the input. However, it seems to handle the apostrophe in both input and patterns. Explain.
-  **Exercise 5.5 [h]** Alter the input mechanism to handle commas and other punctuation characters. Also arrange so that the user doesn't have to type parentheses around the whole input expression. (Hint: this can only be done using some Lisp functions we have not seen yet. Look at `read-line` and `read-from-string`.)

-  **Exercise 5.6 [m]** Modify ELIZA to have an explicit exit. Also arrange so that the output is not printed in parentheses either.
-  **Exercise 5.7 [m]** Add the “memory mechanism” discussed previously to ELIZA. Also add some way of defining synonyms like “everyone” and “everybody.”
-  **Exercise 5.8 [h]** It turns out that none of the rules in the given script uses a variable more than once—there is no rule of the form  $(?x \dots ?x)$ . Write a pattern matcher that only adds bindings, never checks variables against previous bindings. Use the `time` special form to compare your function against the current version.
-  **Exercise 5.9 [h]** Winston and Horn’s book *Lisp* presents a good pattern-matching program. Compare their implementation with this one. One difference is that they handle the case where the first element of the pattern is a segment variable with the following code (translated into our notation):

```
(or (pat-match (rest pattern) (rest input) bindings)
            (pat-match pattern (rest input) bindings))
```

This says that a segment variable matches either by matching the first element of the input, or by matching more than the first element. It is much simpler than our approach using position, partly because they don’t update the binding list. Can you change their code to handle bindings, and incorporate it into our version of `pat-match`? Is it still simpler? Is it more or less efficient?

-  **Exercise 5.10** What is wrong with the following definition of `simple-equal`?

```
(defun simple-equal (x y)
  "Test if two lists or atoms are equal."
  :: Warning - incorrect
  (or (eql x y)
      (and (listp x) (listp y)
           (simple-equal (first x) (first y))
           (simple-equal (rest x) (rest y)))))
```

-  **Exercise 5.11 [m]** Weigh the advantages of changing `no-bindings` to `nil`, and `fail` to something else.

- ❑ **Exercise 5.12 [m]** Weigh the advantages of making `pat-match` return multiple values: the first would be true for a match and false for failure, and the second would be the binding list.
- ❑ **Exercise 5.13 [m]** Suppose that there is a call to `segment-match` where the variable already has a binding. The current definition will keep making recursive calls to `segment-match`, one for each possible matching position. But this is silly—if the variable is already bound, there is only one sequence that it can possibly match against. Change the definition so that it looks only for this one sequence.
- ❑ **Exercise 5.14 [m]** Define a version of `mappend` that, like `mapcar`, accepts any number of argument lists.
- ❑ **Exercise 5.15 [m]** Give an informal proof that `segment-match` always terminates.
- ❑ **Exercise 5.16 [s]** Trick question: There is an object in Lisp which, when passed to `variable-p`, results in an error. What is that object?
- ❑ **Exercise 5.17 [m]** The current version of ELIZA takes an input, transforms it according to the first applicable rule, and outputs the result. One can also imagine a system where the input might be transformed several times before the final output is printed. Would such a system be more powerful? If so, in what way?
- ❑ **Exercise 5.18 [h]** Read Weizenbaum’s original article on ELIZA and transpose his list of rules into the notation used in this chapter.

## 5.7 Answers

**Answer 5.1** No. If either the pattern or the input were shorter, but matched every existing element, the every expression would incorrectly return true.

```
(every #'pat-match '(a b c) '(a)) ⇒ T
```

Furthermore, if either the pattern or the input were a dotted list, then the result of the every would be undefined—some implementations might signal an error, and others might just ignore the expression after the dot.

```
(every #'pat-match '(a b . c) '(a b . d)) ⇒ T, NIL, or error.
```

**Answer 5.4** The expression don't may look like a single word, but to the Lisp reader it is composed of the two elements don and 't, or (quote t). If these elements are used consistently, they will match correctly, but they won't print quite right—there will be a space before the quote mark. In fact the :pretty t argument to write is specified primarily to make (quote t) print as 't (See page 559 of Steele's *Common Lisp the Language*, 2d edition.)

**Answer 5.5** One way to do this is to read a whole line of text with read-line rather than read. Then, substitute spaces for any punctuation character in that string. Finally, wrap the string in parentheses, and read it back in as a list:

```
(defun read-line-no-punct ()
  "Read an input line, ignoring punctuation."
  (read-from-string
    (concatenate 'string "(" (substitute-if #\space #'punctuation-p
                                              (read-line))
                  ")")))
(defun punctuation-p (char) (find char ",;:'!?#-()\\\"")
```

This could also be done by altering the readtable, as in section 23.5, page 821.

**Answer 5.6**

```
(defun eliza ()
  "Respond to user input using pattern matching rules."
  (loop
    (print 'eliza>)
    (let* ((input (read-line-no-punct))
           (response (flatten (use-eliza-rules input))))
      (print-with-spaces response)
      (if (equal response '(good bye)) (RETURN)))))

(defun print-with-spaces (list)
  (mapc #'(lambda (x) (prin1 x) (princ " ")) list))
or
(defun print-with-spaces (list)
  (format t "~{~a ~}" list))
```

**Answer 5.10** Hint: consider (simple-equal '() '(nil . nil)).

**Answer 5.14**

```
(defun mappend (fn &rest list)
  "Apply fn to each element of lists and append the results."
  (apply #'append (apply #'mapcar fn lists)))
```

**Answer 5.16** It must be a symbol, because for nonsymbols, variable-p just returns nil. Getting the symbol-name of a symbol is just accessing a slot, so that can't cause an error. The only thing left is elt; if the symbol name is the empty string, then accessing element zero of the empty string is an error. Indeed, there is a symbol whose name is the empty string: the symbol .

**Answer 5.17** Among other things, a recursive transformation system could be used to handle abbreviations. That is, a form like "don't" could be transformed into "do not" and then processed again. That way, the other rules need only work on inputs matching "do not."

**Answer 5.19** The following includes most of Weizenbaum's rules:

```
(defparameter *eliza-rules*
  '(((?* ?x) hello (?* ?y))
    (How do you do. Please state your problem.))
  (((?* ?x) computer (?* ?y))
    (Do computers worry you?) (What do you think about machines?))
  (((?* ?x) name (?* ?y))
    (Why do you mention computers?))
  (((?* ?x) sorry (?* ?y))
    (What do you think machines have to do with your problem?))
  (((?* ?x) I remember (?* ?y))
    (I am not interested in names))
  (((?* ?x) if (?* ?y))
    (Please don't apologize) (Apologies are not necessary)
    (What feelings do you have when you apologize))
  (((?* ?x) dream (?* ?y))
    (Do you often think of ?y)
    (Does thinking of ?y bring anything else to mind?))
  (((?* ?x) dreamt (?* ?y))
    (What else do you remember) (Why do you recall ?y right now?))
  (((?* ?x) dreamt (?* ?y))
    (What in the present situation reminds you of ?y)
    (What is the connection between me and ?y))
  (((?* ?x) dreamt (?* ?y))
    (Do you really think its likely that ?y) (Do you wish that ?y)
    (What do you think about ?y) (Really-- if ?y))
  (((?* ?x) dreamt (?* ?y))
    (Really-- ?y) (Have you ever fantasized ?y while you were awake?))
  (((?* ?x) dreamt (?* ?y))
    (Have you dreamt ?y before?))
  (((?* ?x) dreamt (?* ?y))
    (How do you feel about ?y in reality?))
  (((?* ?x) dreamt (?* ?y))
    (What does this dream suggest to you?) (Do you dream often?))
  (((?* ?x) dreamt (?* ?y))
    (What persons appear in your dreams?))
  (((?* ?x) dreamt (?* ?y))
    (Don't you believe that dream has to do with your problem?))
  (((?* ?x) my mother (?* ?y))
    (Who else in your family ?y) (Tell me more about your family))
  (((?* ?x) my father (?* ?y))
    (Your father) (Does he influence you strongly?))
  (((?* ?x) my father (?* ?y))
    (What else comes to mind when you think of your father?))
```

(((\*?x) I want (\*?y))  
(What would it mean if you got ?y)  
(Why do you want ?y) (Suppose you got ?y soon))  
(((?\*?x) I am glad (\*?y))  
(How have I helped you to be ?y) (What makes you happy just now)  
(Can you explain why you are suddenly ?y))  
(((?\*?x) I am sad (\*?y))  
(I am sorry to hear you are depressed)  
(I'm sure it's not pleasant to be sad))  
(((?\*?x) are like (\*?y))  
(What resemblance do you see between ?x and ?y))  
(((?\*?x) is like (\*?y))  
(In what way is it that ?x is like ?y)  
(What resemblance do you see?)  
(Could there really be some connection?) (How?))  
(((?\*?x) alike (\*?y))  
(In what way?) (What similarities are there?))  
(((?\*?x) same (\*?y))  
(What other connections do you see?))

(((\*?x) I was (\*?y))  
(Were you really?) (Perhaps I already knew you were ?y)  
(Why do you tell me you were ?y now?))  
(((?\*?x) was I (\*?y))  
(What if you were ?y ?) (Do you think you were ?y)  
(What would it mean if you were ?y))  
(((?\*?x) I am (\*?y))  
(In what way are you ?y) (Do you want to be ?y ?))  
(((?\*?x) am I (\*?y))  
(Do you believe you are ?y) (Would you want to be ?y)  
(You wish I would tell you you are ?y)  
(What would it mean if you were ?y))  
(((?\*?x) am (\*?y))  
(Why do you say "AM?") (I don't understand that))  
(((?\*?x) are you (\*?y))  
(Why are you interested in whether I am ?y or not?)  
(Would you prefer if I weren't ?y)  
(Perhaps I am ?y in your fantasies))  
(((?\*?x) you are (\*?y))  
(What makes you think I am ?y ?))

(((\*? x) because (\*? y))  
(Is that the real reason?) (What other reasons might there be?)  
(Does that reason seem to explain anything else?))  
(((?\* ?x) were you (\*? y))  
(Perhaps I was ?y) (What do you think?) (What if I had been ?y))  
(((?\* ?x) I can't (\*? y))  
(Maybe you could ?y now) (What if you could ?y ?))  
(((?\* ?x) I feel (\*? y))  
(Do you often feel ?y ?))  
(((?\* ?x) I felt (\*? y))  
(What other feelings do you have?))  
(((?\* ?x) I (\*? y) you (\*? ?z))  
(Perhaps in your fantasy we ?y each other))  
(((?\* ?x) why don't you (\*? y))  
(Should you ?y yourself?)  
(Do you believe I don't ?y) (Perhaps I will ?y in good time))  
(((?\* ?x) yes (\*? y))  
(You seem quite positive) (You are sure) (I understand))  
(((?\* ?x) no (\*? y))  
(Why not?) (You are being a bit negative)  
(Are you saying "NO" just to be negative?))  
(((?\* ?x) someone (\*? y))  
(Can you be more specific?))  
(((?\* ?x) everyone (\*? y))  
(surely not everyone) (Can you think of anyone in particular?)  
(Who for example?) (You are thinking of a special person))  
(((?\* ?x) always (\*? y))  
(Can you think of a specific example) (When?)  
(What incident are you thinking of?) (Really-- always))  
(((?\* ?x) what (\*? y))  
(Why do you ask?) (Does that question interest you?)  
(What is it you really want to know?) (What do you think?)  
(What comes to your mind when you ask that?))  
(((?\* ?x) perhaps (\*? y))  
(You do not seem quite certain))  
(((?\* ?x) are (\*? y))  
(Did you think they might not be ?y)  
(Possibly they are ?y))  
(((?\* ?x))  
(Very interesting) (I am not sure I understand you fully)  
(What does that suggest to you?) (Please continue) (Go on)  
(Do you feel strongly about discussing such things?)))

## *CHAPTER* 6

---

# Building Software Tools

*Man is a tool-using animal. . . .  
Without tools he is nothing,  
with tools he is all.*  
—Thomas Carlyle (1795–1881)

In chapters 4 and 5 we were concerned with building two particular programs, GPS and ELIZA. In this chapter, we will reexamine those two programs to discover some common patterns. Those patterns will be abstracted out to form reusable software tools that will prove helpful in subsequent chapters.

### 6.1 An Interactive Interpreter Tool

The structure of the function `eliza` is a common one. It is repeated below:

```
(defun eliza ()  
  "Respond to user input using pattern matching rules."  
  (loop  
    (print 'eliza>)  
    (print (flatten (use-eliza-rules (read))))))
```

Many other applications use this pattern, including Lisp itself. The top level of Lisp could be defined as:

```
(defun lisp ()
  (loop
    (print '>)
    (print (eval (read)))))
```

The top level of a Lisp system has historically been called the “read-eval-print loop.” Most modern Lisps print a prompt before reading input, so it should really be called the “prompt-read-eval-print loop,” but there was no prompt in some early systems like MacLisp, so the shorter name stuck. If we left out the prompt, we could write a complete Lisp interpreter using just four symbols:

```
(loop (print (eval (read))))
```

It may seem facetious to say those four symbols and eight parentheses constitute a Lisp interpreter. When we write that line, have we really accomplished anything? One answer to that question is to consider what we would have to do to write a Lisp (or Pascal) interpreter in Pascal. We would need a lexical analyzer and a symbol table manager. This is a considerable amount of work, but it is all handled by `read`. We would need a syntactic parser to assemble the lexical tokens into statements. `read` also handles this, but only because Lisp statements have trivial syntax: the syntax of lists and atoms. Thus `read` serves fine as a syntactic parser for Lisp, but would fail for Pascal. Next, we need the evaluation or interpretation part of the interpreter; `eval` does this nicely, and could handle Pascal just as well if we parsed Pascal syntax into Lisp expressions. `print` does much less work than `read` or `eval`, but is still quite handy.

The important point is not whether one line of code can be considered an implementation of Lisp; it is to recognize common patterns of computation. Both `eliza` and `lisp` can be seen as interactive interpreters that read some input, transform or evaluate the input in some way, print the result, and then go back for more input. We can extract the following common pattern:

```
(defun program ()
  (loop
    (print prompt)
    (print (transform (read)))))
```

There are two ways to make use of recurring patterns like this: formally and informally. The informal alternative is to treat the pattern as a cliche or idiom that will occur frequently in our writing of programs but will vary from use to use. When we

want to write a new program, we remember writing or reading a similar one, go back and look at the first program, copy the relevant sections, and then modify them for the new program. If the borrowing is extensive, it would be good practice to insert a comment in the new program citing the original, but there would be no “official” connection between the original and the derived program.

The formal alternative is to create an abstraction, in the form of functions and perhaps data structures, and refer explicitly to that abstraction in each new application—in other words, to capture the abstraction in the form of a useable software tool. The interpreter pattern could be abstracted into a function as follows:

```
(defun interactive-interpreter (prompt transformer)
  "Read an expression, transform it, and print the result."
  (loop
    (print prompt)
    (print (funcall transformer (read)))))
```

This function could then be used in writing each new interpreter:

```
(defun lisp ()
  (interactive-interpreter 'gt #'eval))

(defun eliza ()
  (interactive-interpreter 'eliza>
    #'(lambda (x) (flatten (use-eliza-rules x)))))
```

Or, with the help of the higher-order function compose:

```
(defun compose (f g)
  "Return the function that computes (f (g x))."
  #'(lambda (x) (funcall f (funcall g x)))))

(defun eliza ()
  (interactive-interpreter 'eliza>
    (compose #'flatten #'use-eliza-rules)))
```

There are two differences between the formal and informal approaches. First, they look different. If the abstraction is a simple one, as this one is, then it is probably easier to read an expression that has the loop explicitly written out than to read one that calls `interactive-interpreter`, since that requires finding the definition of `interactive-interpreter` and understanding it as well.

The other difference shows up in what’s called *maintenance*. Suppose we find a missing feature in the definition of the interactive interpreter. One such omission is that the loop has no exit. I have been assuming that the user can terminate the loop by hitting some interrupt (or break, or abort) key. A cleaner implementation would allow

the user to give the interpreter an explicit termination command. Another useful feature would be to handle errors within the interpreter. If we use the informal approach, then adding such a feature to one program would have no effect on the others. But if we use the formal approach, then improving interactive-interpreter would automatically bring the new features to all the programs that use it.

The following version of interactive-interpreter adds two new features. First, it uses the macro `handler-case`<sup>1</sup> to handle errors. This macro evaluates its first argument, and normally just returns that value. However, if an error occurs, the subsequent arguments are checked for an error condition that matches the error that occurred. In this use, the case `error` matches all errors, and the action taken is to print the error condition and continue.

This version also allows the prompt to be either a string or a function of no arguments that will be called to print the prompt. The function `prompt-generator`, for example, returns a function that will print prompts of the form [1], [2], and so forth.

```
(defun interactive-interpreter (prompt transformer)
  "Read an expression, transform it, and print the result."
  (loop
    (handler-case
      (progn
        (if (stringp prompt)
            (print prompt)
            (funcall prompt))
        (print (funcall transformer (read))))
      ;; In case of error, do this:
      (error (condition)
        (format t "~&; Error ~a ignored, back to top level."
                condition)))))

(defun prompt-generator (&optional (num 0) (ctl-string "[~d] "))
  "Return a function that prints prompts like [1], [2], etc."
  #'(lambda () (format t ctl-string (incf num))))
```

## 6.2 A Pattern-Matching Tool

The `pat-match` function was a pattern matcher defined specifically for the ELIZA program. Subsequent programs will need pattern matchers too, and rather than write specialized matchers for each new program, it is easier to define one general

---

<sup>1</sup>The macro `handler-case` is only in ANSI Common Lisp.

pattern matcher that can serve most needs, and is extensible in case novel needs come up.

The problem in designing a “general” tool is deciding what features to provide. We can try to define features that might be useful, but it is also a good idea to make the list of features open-ended, so that new ones can be easily added when needed.

Features can be added by generalizing or specializing existing ones. For example, we provide segment variables that match zero or more input elements. We can specialize this by providing for a kind of segment variable that matches one or more elements, or for an optional variable that matches zero or one element. Another possibility is to generalize segment variables to specify a match of  $m$  to  $n$  elements, for any specified  $m$  and  $n$ . These ideas come from experience with notations for writing regular expressions, as well as from very general heuristics for generalization, such as “consider important special cases” and “zero and one are likely to be important special cases.”

Another useful feature is to allow the user to specify an arbitrary predicate that a match must satisfy. The notation (`?is ?n numberp`) could be used to match any expression that is a number and bind it to the variable `?n`. This would look like:

```
> (pat-match '(x = (?is ?n numberp)) '(x = 34)) => ((?n . 34))
> (pat-match '(x = (?is ?n numberp)) '(x = x)) => NIL
```

Since patterns are like boolean expressions, it makes sense to allow boolean operators on them. Following the question-mark convention, we will use `?and`, `?or` and `?not` for the operators.<sup>2</sup> Here is a pattern to match a relational expression with one of three relations. It succeeds because the `<` matches one of the three possibilities specified by (`?or <=>`).

```
> (pat-match '(%x (?or < = >) %y) '(3 < 4)) => ((?Y . 4) (%X . 3))
```

Here is an example of an `?and` pattern that checks if an expression is both a number and odd:

```
> (pat-match '(x = (?and (?is ?n numberp) (?is ?n oddp)))
              '(x = 3))
((?N . 3))
```

---

<sup>2</sup>An alternative would be to reserve the question mark for variables only and use another notation for these match operators. Keywords would be a good choice, such as `:and`, `:or`, `:is`, etc.

The next pattern uses `?not` to insure that two parts are not equal:

```
> (pat-match '(?x /= (?not ?x)) '(3 /= 4)) => ((?X . 3))
```

The segment matching notation we have seen before. It is augmented to allow for three possibilities: zero or more expressions; one or more expressions; and zero or one expressions. Finally, the notation `(?if exp)` can be used to test a relationship between several variables. It has to be listed as a segment pattern rather than a single pattern because it does not consume any of the input at all:

```
> (pat-match '(?x > ?y (?if (> ?x ?y))) '(4 > 3)) =>
((?Y . 3) (?X . 4))
```

When the description of a problem gets this complicated, it is a good idea to attempt a more formal specification. The following table describes a grammar of patterns, using the same grammar rule format described in chapter 2.

$pat \Rightarrow$	$var$	match any one expression
	$constant$	match just this atom
	$segment-pat$	match something against a sequence
	$single-pat$	match something against one expression
	$(pat . pat)$	match the first and the rest
$single-pat \Rightarrow$	$(?is var predicate)$	test predicate on one expression
	$(?or pat...)$	match any pattern on one expression
	$(?and pat...)$	match every pattern on one expression
	$(?not pat...)$	succeed if pattern(s) do not match
$segment-pat \Rightarrow$	$((?* var) ...)$	match zero or more expressions
	$((?+ var) ...)$	match one or more expressions
	$((?? var) ...)$	match zero or one expression
	$((?if exp) ...)$	test if exp (which may contain variables) is true
$var \Rightarrow$	$?chars$	a symbol starting with <code>?</code>
$constant \Rightarrow$	$atom$	any nonvariable atom

Despite the added complexity, all patterns can still be classified into five cases. The pattern must be either a variable, constant, a (generalized) segment pattern, a (generalized) single-element pattern, or a cons of two patterns. The following definition of `pat-match` reflects the five cases (along with two checks for failure):

```
(defun pat-match (pattern input &optional (bindings no-bindings))
  "Match pattern against input in the context of the bindings"
  (cond ((eq bindings fail) fail)
        ((variable-p pattern)
         (match-variable pattern input bindings))
        ((eql pattern input) bindings)
        ((segment-pattern-p pattern)
         (segment-matcher pattern input bindings))
        ((single-pattern-p pattern) ; ***
         (single-matcher pattern input bindings)) ; ***
        ((and (consp pattern) (consp input))
         (pat-match (rest pattern) (rest input)
                    (pat-match (first pattern) (first input)
                               bindings)))
        (t fail)))
```

For completeness, we repeat here the necessary constants and low-level functions from ELIZA:

```
(defconstant fail nil "Indicates pat-match failure")

(defconstant no-bindings '((t . t))
  "Indicates pat-match success, with no variables.")

(defun variable-p (x)
  "Is x a variable (a symbol beginning with '?')?"
  (and (symbolp x) (equal (char (symbol-name x) 0) #\?)))

(defun get-binding (var bindings)
  "Find a (variable . value) pair in a binding list."
  (assoc var bindings))

(defun binding-var (binding)
  "Get the variable part of a single binding."
  (car binding))

(defun binding-val (binding)
  "Get the value part of a single binding."
  (cdr binding))

(defun make-binding (var val) (cons var val))

(defun lookup (var bindings)
  "Get the value part (for var) from a binding list."
  (binding-val (get-binding var bindings)))
```

```
(defun extend-bindings (var val bindings)
  "Add a (var . value) pair to a binding list."
  (cons (make-binding var val)
        ;; Once we add a "real" binding,
        ;; we can get rid of the dummy no-bindings
        (if (eq bindings no-bindings)
            nil
            bindings))

(defun match-variable (var input bindings)
  "Does VAR match input?  Uses (or updates) and returns bindings."
  (let ((binding (get-binding var bindings)))
    (cond ((not binding) (extend-bindings var input bindings))
          ((equal input (binding-val binding)) bindings)
          (t fail))))
```

The next step is to define the predicates that recognize generalized segment and single-element patterns, and the matching functions that operate on them. We could implement `segment-matcher` and `single-matcher` with `case` statements that consider all possible cases. However, that would make it difficult to extend the matcher. A programmer who wanted to add a new kind of segment pattern would have to edit the definitions of both `segment-pattern-p` and `segment-matcher` to install the new feature. This by itself may not be too bad, but consider what happens when two programmers each add independent features. If you want to use both, then neither version of `segment-matcher` (or `segment-pattern-p`) will do. You'll have to edit the functions again, just to merge the two extensions.

The solution to this dilemma is to write one version of `segment-pattern-p` and `segment-matcher`, once and for all, but to have these functions refer to a table of pattern/action pairs. The table would say "if you see `?*` in the pattern, then use the function `segment-match`," and so on. Then programmers who want to extend the matcher just add entries to the table, and it is trivial to merge different extensions (unless of course two programmers have chosen the same symbol to mark different actions).

This style of programming, where pattern/action pairs are stored in a table, is called *data-driven programming*. It is a very flexible style that is appropriate for writing extensible systems.

There are many ways to implement tables in Common Lisp, as discussed in section 3.6, page 73. In this case, the keys to the table will be symbols (like `?*`), and it is fine if the representation of the table is distributed across memory. Thus, property lists are an appropriate choice. We will have two tables, represented by the `segment-match` property and the `single-match` property of symbols like `?*`. The value of each property will be the name of a function that implements the match. Here are the table entries to implement the grammar listed previously:

```
(setf (get '?is 'single-match) 'match-is)
(setf (get '?or 'single-match) 'match-or)
(setf (get '?and 'single-match) 'match-and)
(setf (get '?not 'single-match) 'match-not)

(setf (get '?* 'segment-match) 'segment-match)
(setf (get '?+ 'segment-match) 'segment-match+)
(setf (get '?? 'segment-match) 'segment-match?)
(setf (get '?if 'segment-match) 'match-if)
```

With the table defined, we need to do two things. First, define the “glue” that holds the table together: the predicates and action-taking functions. A function that looks up a data-driven function and calls it (such as `segment-matcher` and `single-matcher`) is called a *dispatch function*.

```
(defun segment-pattern-p (pattern)
  "Is this a segment-matching pattern like ((?* var) . pat)?"
  (and (consp pattern) (consp (first pattern))
        (symbolp (first (first pattern)))
        (segment-match-fn (first (first pattern)))))

(defun single-pattern-p (pattern)
  "Is this a single-matching pattern?
  E.g. (?is x predicate) (?and . patterns) (?or . patterns)."
  (and (consp pattern)
        (single-match-fn (first pattern)))))

(defun segment-matcher (pattern input bindings)
  "Call the right function for this kind of segment pattern."
  (funcall (segment-match-fn (first (first pattern)))
           pattern input bindings))

(defun single-matcher (pattern input bindings)
  "Call the right function for this kind of single pattern."
  (funcall (single-match-fn (first pattern))
           (rest pattern) input bindings))

(defun segment-match-fn (x)
  "Get the segment-match function for x,
  if it is a symbol that has one."
  (when (symbolp x) (get x 'segment-match)))

(defun single-match-fn (x)
  "Get the single-match function for x,
  if it is a symbol that has one."
  (when (symbolp x) (get x 'single-match)))
```

The last thing to do is define the individual matching functions. First, the single-pattern matching functions:

```
(defun match-is (var-and-pred input bindings)
  "Succeed and bind var if the input satisfies pred,
  where var-and-pred is the list (var pred)."
  (let* ((var (first var-and-pred))
         (pred (second var-and-pred))
         (new-bindings (pat-match var input bindings)))
    (if (or (eq new-bindings fail)
             (not (funcall pred input)))
        fail
        new-bindings)))

(defun match-and (patterns input bindings)
  "Succeed if all the patterns match the input."
  (cond ((eq bindings fail) fail)
        ((null patterns) bindings)
        (t (match-and (rest patterns) input
                      (pat-match (first patterns) input
                                 bindings)))))

(defun match-or (patterns input bindings)
  "Succeed if any one of the patterns match the input."
  (if (null patterns)
      fail
      (let ((new-bindings (pat-match (first patterns)
                                     input bindings)))
        (if (eq new-bindings fail)
            (match-or (rest patterns) input bindings)
            new-bindings)))

(defun match-not (patterns input bindings)
  "Succeed if none of the patterns match the input.
  This will never bind any variables."
  (if (match-or patterns input bindings)
      fail
      bindings))
```

Now the segment-pattern matching functions. `segment-match` is similar to the version presented as part of ELIZA. The difference is in how we determine `pos`, the position of the first element of the input that could match the next element of the pattern after the `segment` variable. In ELIZA, we assumed that the `segment` variable was either the last element of the pattern or was followed by a constant. In the following version, we allow nonconstant patterns to follow `segment` variables. The function `first-match-pos` is added to handle this. If the following element is in fact a constant, the same calculation is done using `position`. If it is not a constant, then

we just return the first possible starting position—unless that would put us past the end of the input, in which case we return nil to indicate failure:

```
(defun segment-match (pattern input bindings &optional (start 0))
  "Match the segment pattern ((?* var) . pat) against input."
  (let ((var (second (first pattern)))
        (pat (rest pattern)))
    (if (null pat)
        (match-variable var input bindings)
        (let ((pos (first-match-pos (first pat) input start)))
          (if (null pos)
              fail
              (let ((b2 (pat-match
                         pat (subseq input pos)
                         (match-variable var (subseq input 0 pos)
                                         bindings))))
                ;; If this match failed, try another longer one
                (if (eq b2 fail)
                    (segment-match pattern input bindings (+ pos 1))
                    b2))))))

(defun first-match-pos (pat1 input start)
  "Find the first position that pat1 could possibly match input,
  starting at position start.  If pat1 is non-constant, then just
  return start."
  (cond ((and (atom pat1) (not (variable-p pat1)))
         (position pat1 input :start start :test #'equal))
        ((< start (length input)) start)
        (t nil)))
```

In the first example below, the segment variable ?x matches the sequence (b c). In the second example, there are two segment variables in a row. The first successful match is achieved with the first variable, ?x, matching the empty sequence, and the second one, ?y, matching (b c).

```
> (pat-match '(a (?* ?x) d) '(a b c d)) => ((?X B C))
> (pat-match '(a (?* ?x) (?* ?y) d) '(a b c d)) => ((?Y B C) (?X))
```

In the next example, ?x is first matched against nil and ?y against (b c d), but that fails, so we try matching ?x against a segment of length one. That fails too, but finally the match succeeds with ?x matching the two-element segment (b c), and ?y matching (d).

```
> (pat-match '(a (?* ?x) (?* ?y) ?x ?y)
             '(a b c d (b c) (d))) => ((?Y D) (?X B C))
```

Given `segment-match`, it is easy to define the function to match one-or-more elements and the function to match zero-or-one element:

```
(defun segment-match+ (pattern input bindings)
  "Match one or more elements of input."
  (segment-match pattern input bindings 1))

(defun segment-match? (pattern input bindings)
  "Match zero or one element of input."
  (let ((var (second (first pattern)))
        (pat (rest pattern)))
    (or (pat-match (cons var pat) input bindings)
        (pat-match pat input bindings))))
```

Finally, we supply the function to test an arbitrary piece of Lisp code. It does this by evaluating the code with the bindings implied by the binding list. This is one of the few cases where it is appropriate to call `eval`: when we want to give the user unrestricted access to the Lisp interpreter.

```
(defun match-if (pattern input bindings)
  "Test an arbitrary expression involving variables.
  The pattern looks like ((?if code) . rest)."
  (and (progv (mapcar #'car bindings)
               (mapcar #'cdr bindings)
               (eval (second (first pattern))))
       (pat-match (rest pattern) input bindings)))
```

Here are two examples using `?if`. The first succeeds because `(+ 3 4)` is indeed 7, and the second fails because `(> 3 4)` is false.

```
> (pat-match '(%x ?op ?y is ?z (%if (= op ?x ?y) ?z))
              '(3 + 4 is 7))
   ((?Z . 7) (?Y . 4) (?OP . +) (?X . 3))

> (pat-match '(%x ?op ?y (%if (= op ?x ?y)))
              '(3 > 4))
   NIL
```

The syntax we have defined for patterns has two virtues: first, the syntax is very general, so it is easy to extend. Second, the syntax can be easily manipulated by `pat-match`. However, there is one drawback: the syntax is a little verbose, and some may find it ugly. Compare the following two patterns:

```
(a (?* ?x) (?* ?y) d)
(a ?x* ?y* d)
```

Many readers find the second pattern easier to understand at a glance. We could change pat-match to allow for patterns of the form  $?x^*$ , but that would mean pat-match would have a lot more work to do on every match. An alternative is to leave pat-match as is, but define another level of syntax for use by human readers only. That is, a programmer could type the second expression above, and have it translated into the first, which would then be processed by pat-match.

In other words, we will define a facility to define a kind of pattern-matching macro that will be expanded the first time the pattern is seen. It is better to do this expansion once than to complicate pat-match and in effect do the expansion every time a pattern is used. (Of course, if a pattern is only used once, then there is no advantage. But in most programs, each pattern will be used again and again.)

We need to define two functions: one to define pattern-matching macros, and another to expand patterns that may contain these macros. We will only allow symbols to be macros, so it is reasonable to store the expansions on each symbol's property list:

```
(defun pat-match-abbrev (symbol expansion)
  "Define symbol as a macro standing for a pat-match pattern."
  (setf (get symbol 'expand-pat-match-abbrev)
        (expand-pat-match-abbrev expansion)))

(defun expand-pat-match-abbrev (pat)
  "Expand out all pattern matching abbreviations in pat."
  (cond ((and (symbolp pat) (get pat 'expand-pat-match-abbrev)))
         ((atom pat) pat)
         (t (cons (expand-pat-match-abbrev (first pat))
                  (expand-pat-match-abbrev (rest pat))))))
```

We would use this facility as follows:

```
> (pat-match-abbrev '?x* '(*? ?x)) => (*? ?X)
> (pat-match-abbrev '?y* '(*? ?y)) => (*? ?Y)
> (setf axyd (expand-pat-match-abbrev '(a ?x* ?y* d))) =>
(A (*? ?X) (*? ?Y) D)
> (pat-match axyd '(a b c d)) => ((?Y B C) (?X))
```



**Exercise 6.1[m]** Go back and change the ELIZA rules to use the abbreviation facility. Does this make the rules easier to read?



**Exercise 6.2 [h]** In the few prior examples, every time there was a binding of pattern variables that satisfied the input, that binding was found. Informally, show that `pat-match` will always find such a binding, or show a counterexample where it fails to find one.

## 6.3 A Rule-Based Translator Tool

As we have defined it, the pattern matcher matches one input against one pattern. In `eliza`, we need to match each input against a number of patterns, and then return a result based on the rule that contains the first pattern that matches. To refresh your memory, here is the function `use-eliza-rules`:

```
(defun use-eliza-rules (input)
  "Find some rule with which to transform the input."
  (some #'(lambda (rule)
            (let ((result (pat-match (rule-pattern rule) input)))
              (if (not (eq result fail))
                  (sublis (switch-viewpoint result)
                          (random-elt (rule-responses rule)))))))
  *eliza-rules*))
```

It turns out that this will be a quite common thing to do: search through a list of rules for one that matches, and take action according to that rule. To turn the structure of `use-eliza-rules` into a software tool, we will allow the user to specify each of the following:

- What kind of rule to use. Every rule will be characterized by an if-part and a then-part, but the ways of getting at those two parts may vary.
- What list of rules to use. In general, each application will have its own list of rules.
- How to see if a rule matches. By default, we will use `pat-match`, but it should be possible to use other matchers.
- What to do when a rule matches. Once we have determined which rule to use, we have to determine what it means to use it. The default is just to substitute the bindings of the match into the then-part of the rule.

The rule-based translator tool now looks like this:

```
(defun rule-based-translator
  (input rules &key (matcher #'pat-match)
        (rule-if #'first) (rule-then #'rest) (action #'sublis))
  "Find the first rule in rules that matches input,
  and apply the action to that rule."
  (some
   #'(lambda (rule)
       (let ((result (funcall matcher (funcall rule-if rule)
                               input)))
         (if (not (eq result fail))
             (funcall action result (funcall rule-then rule))))))
  rules))

(defun use-eliza-rules (input)
  "Find some rule with which to transform the input."
  (rule-based-translator input *eliza-rules*
    :action #'(lambda (bindings responses)
                (sublis (switch-viewpoint bindings)
                        (random-elt responses)))))
```

## 6.4 A Set of Searching Tools

The GPS program can be seen as a problem in *search*. In general, a search problem involves exploring from some starting state and investigating neighboring states until a solution is reached. As in GPS, *state* means a description of any situation or state of affairs. Each state may have several neighbors, so there will be a choice of how to search. We can travel down one path until we see it is a dead end, or we can consider lots of different paths at the same time, expanding each path step by step. Search problems are called *nondeterministic* because there is no way to determine what is the best step to take next. AI problems, by their very nature, tend to be nondeterministic. This can be a source of confusion for programmers who are used to deterministic problems. In this section we will try to clear up that confusion. This section also serves as an example of how higher-order functions can be used to implement general tools that can be specified by passing in specific functions.

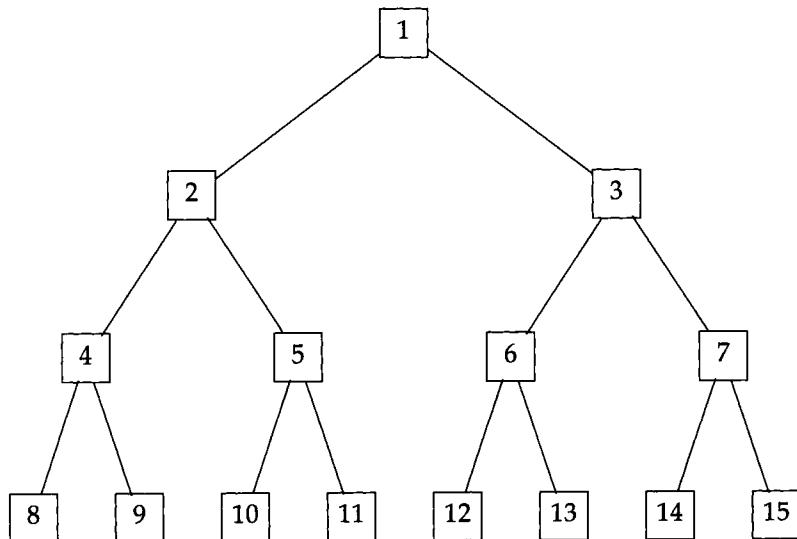
Abstractly, a search problem can be characterized by four features:

- The *start* state.
- The *goal* state (or states).

- The *successors*, or states that can be reached from any other state.
- The *strategy* that determines the *order* in which we search.

The first three features are part of the problem, while the fourth is part of the solution. In GPS, the starting state was given, along with a description of the goal states. The successors of a state were determined by consulting the operators. The search strategy was means-ends analysis. This was never spelled out explicitly but was implicit in the structure of the whole program. In this section we will formulate a general searching tool, show how it can be used to implement several different search strategies, and then show how GPS could be implemented with this tool.

The first notion we have to define is the *state space*, or set of all possible states. We can view the states as nodes and the successor relation as links in a graph. Some state space graphs will have a small number of states, while others have an infinite number, but they can still be solved if we search cleverly. Some graphs will have a regular structure, while others will appear random. We will start by considering only trees—that is, graphs where a state can be reached by only one unique sequence of successor links. Here is a tree:



### *Searching Trees*

We will call our first searching tool *tree-search*, because it is designed to search state spaces that are in the form of trees. It takes four arguments: (1) a list of valid starting states, (2) a predicate to decide if we have reached a goal state, (3) a function to generate the successors of a state, and (4) a function that decides in what order

to search. The first argument is a list rather than a single state so that tree-search can recursively call itself after it has explored several paths through the state space. Think of the first argument not as a starting state but as a list of possible states from which the goal may be reached. This lists represents the fringe of the tree that has been explored so far. tree-search has three cases: If there are no more states to consider, then give up and return fail. If the first possible state is a goal state, then return the successful state. Otherwise, generate the successors of the first state and combine them with the other states. Order this combined list according to the particular search strategy and continue searching. Note that tree-search itself does not specify any particular searching strategy.

```
(defun tree-search (states goal-p successors combiner)
  "Find a state that satisfies goal-p. Start with states,
  and search according to successors and combiner."
  (dbg :search "~&;; Search: ~a" states)
  (cond ((null states) fail)
        ((funcall goal-p (first states)) (first states))
        (t (tree-search
            (funcall combiner
              (funcall successors (first states))
              (rest states))
            goal-p successors combiner))))
```

The first strategy we will consider is called *depth-first search*. In depth-first search, the longest paths are considered first. In other words, we generate the successors of a state, and then work on the first successor first. We only return to one of the subsequent successors if we arrive at a state that has no successors at all. This strategy can be implemented by simply appending the previous states to the end of the list of new successors on each iteration. The function depth-first-search takes a single starting state, a goal predicate, and a successor function. It packages the starting state into a list as expected by tree-search, and specifies append as the combining function:

```
(defun depth-first-search (start goal-p successors)
  "Search new states first until goal is reached."
  (tree-search (list start) goal-p successors #'append))
```

Let's see how we can search through the binary tree defined previously. First, we define the successor function `binary-tree`. It returns a list of two states, the two numbers that are twice the input state and one more than twice the input state. So the successors of 1 will be 2 and 3, and the successors of 2 will be 4 and 5. The `binary-tree` function generates an infinite tree of which the first 15 nodes are diagrammed in our example.

```
(defun binary-tree (x) (list (* 2 x) (+ 1 (* 2 x))))
```

To make it easier to specify a goal, we define the function `is` as a function that returns a predicate that tests for a particular value. Note that `is` does not do the test itself. Rather, it returns a function that can be called to perform tests:

```
(defun is (value) #'(lambda (x) (eq x value)))
```

Now we can turn on the debugging output and search through the binary tree, starting at 1, and looking for, say, 12, as the goal state. Each line of debugging output shows the list of states that have been generated as successors but not yet examined:

```
> (debug :search) => (SEARCH)
> (depth-first-search 1 (is 12) #'binary-tree)
;; Search: (1)
;; Search: (2 3)
;; Search: (4 5 3)
;; Search: (8 9 5 3)
;; Search: (16 17 9 5 3)
;; Search: (32 33 17 9 5 3)
;; Search: (64 65 33 17 9 5 3)
;; Search: (128 129 65 33 17 9 5 3)
;; Search: (256 257 129 65 33 17 9 5 3)
;; Search: (512 513 257 129 65 33 17 9 5 3)
;; Search: (1024 1025 513 257 129 65 33 17 9 5 3)
;; Search: (2048 2049 1025 513 257 129 65 33 17 9 5 3)
[Abort]
```

The problem is that we are searching an infinite tree, and the depth-first search strategy just dives down the left-hand branch at every step. The only way to stop the doomed search is to type an interrupt character.

An alternative strategy is *breadth-first search*, where the shortest path is extended first at each step. It can be implemented simply by appending the new successor states to the end of the existing states:

```
(defun prepend (x y) "Prepend y to start of x" (append y x))

(defun breadth-first-search (start goal-p successors)
  "Search old states first until goal is reached."
  (tree-search (list start) goal-p successors #'prepend))
```

The only difference between depth-first and breadth-first search is the difference between `append` and `prepend`. Here we see `breadth-first-search` in action:

```
> (breadth-first-search 1 (is 12) 'binary-tree)
;; Search: (1)
;; Search: (2 3)
;; Search: (3 4 5)
;; Search: (4 5 6 7)
;; Search: (5 6 7 8 9)
;; Search: (6 7 8 9 10 11)
;; Search: (7 8 9 10 11 12 13)
;; Search: (8 9 10 11 12 13 14 15)
;; Search: (9 10 11 12 13 14 15 16 17)
;; Search: (10 11 12 13 14 15 16 17 18 19)
;; Search: (11 12 13 14 15 16 17 18 19 20 21)
;; Search: (12 13 14 15 16 17 18 19 20 21 22 23)
12
```

Breadth-first search ends up searching each node in numerical order, and so it will eventually find any goal. It is methodical, but therefore plodding. Depth-first search will be much faster—if it happens to find the goal at all. For example, if we were looking for 2048, depth-first search would find it in 12 steps, while breadth-first would take 2048 steps. Breadth-first search also requires more storage, because it saves more intermediate states.

If the search tree is finite, then either breadth-first or depth-first will eventually find the goal. Both methods search the entire state space, but in a different order. We will now show a depth-first search of the 15-node binary tree diagrammed previously. It takes about the same amount of time to find the goal (12) as it did with breadth-first search. It would have taken more time to find 15; less to find 8. The big difference is in the number of states considered at one time. At most, depth-first search considers four at a time; in general it will need to store only  $\log_2 n$  states to search a  $n$ -node tree, while breadth-first search needs to store  $n/2$  states.

```
(defun finite-binary-tree (n)
  "Return a successor function that generates a binary tree
  with n nodes."
  #'(lambda (x)
      (remove-if #'(lambda (child) (> child n))
                 (binary-tree x)))))

> (depth-first-search 1 (is 12) (finite-binary-tree 15))
;; Search: (1)
;; Search: (2 3)
;; Search: (4 5 3)
;; Search: (8 9 5 3)
;; Search: (9 5 3)
;; Search: (5 3)
;; Search: (10 11 3)
;; Search: (11 3)
```

```

;; Search: (3)
;; Search: (6 7)
;; Search: (12 13 7)
12

```

### *Guiding the Search*

While breadth-first search is more methodical, neither strategy is able to take advantage of any knowledge about the state space. They both search blindly. In most real applications we will have some estimate of how far a state is from the solution. In such cases, we can implement a *best-first search*. The name is not quite accurate; if we could really search best first, that would not be a search at all. The name refers to the fact that the state that *appears* to be best is searched first.

To implement best-first search we need to add one more piece of information: a cost function that gives an estimate of how far a given state is from the goal.

For the binary tree example, we will use as a cost estimate the numeric difference from the goal. So if we are looking for 12, then 12 has cost 0, 8 has cost 4 and 2048 has cost 2036. The higher-order function `diff`, shown in the following, returns a cost function that computes the difference from a goal. The higher-order function `sorter` takes a cost function as an argument and returns a combiner function that takes the lists of old and new states, appends them together, and sorts the result based on the cost function, lowest cost first. (The built-in function `sort` sorts a list according to a comparison function. In this case the smaller numbers come first. `sort` takes an optional `:key` argument that says how to compute the score for each element. Be careful—`sort` is a destructive function.)

```

(defun diff (num)
  "Return the function that finds the difference from num."
  #'(lambda (x) (abs (- x num)))

(defun sorter (cost-fn)
  "Return a combiner function that sorts according to cost-fn."
  #'(lambda (new old)
      (sort (append new old) #'< :key cost-fn)))

(defun best-first-search (start goal-p successors cost-fn)
  "Search lowest cost states first until goal is reached."
  (tree-search (list start) goal-p successors (sorter cost-fn)))

```

Now, using the difference from the goal as the cost function, we can search using best-first search:

```
> (best-first-search 1 (is 12) #'binary-tree (diff 12))
;; Search: (1)
;; Search: (3 2)
;; Search: (7 6 2)
;; Search: (14 15 6 2)
;; Search: (15 6 2 28 29)
;; Search: (6 2 28 29 30 31)
;; Search: (12 13 2 28 29 30 31)
12
```

The more we know about the state space, the better we can search. For example, if we know that all successors are greater than the states they come from, then we can use a cost function that gives a very high cost for numbers above the goal. The function `price-is-right` is like `diff`, except that it gives a high penalty for going over the goal.<sup>3</sup> Using this cost function leads to a near-optimal search on this example. It makes the “mistake” of searching 7 before 6 (because 7 is closer to 12), but does not waste time searching 14 and 15:

```
(defun price-is-right (price)
  "Return a function that measures the difference from price,
  but gives a big penalty for going over price."
  #'(lambda (x) (if (> x price)
                     most-positive-fixnum
                     (- price x))))
```

```
> (best-first-search 1 (is 12) #'binary-tree (price-is-right 12))
;; Search: (1)
;; Search: (3 2)
;; Search: (7 6 2)
;; Search: (6 2 14 15)
;; Search: (12 2 13 14 15)
12
```

All the searching methods we have seen so far consider ever-increasing lists of states as they search. For problems where there is only one solution, or a small number of solutions, this is unavoidable. To find a needle in a haystack, you need to look at a lot of hay. But for problems with many solutions, it may be worthwhile to discard unpromising paths. This runs the risk of failing to find a solution at all, but it can save enough space and time to offset the risk. A best-first search that keeps only a fixed number of alternative states at any one time is known as a *beam search*. Think of searching as shining a light through the dark of the state space. In other search

---

<sup>3</sup>The built-in constant `most-positive-fixnum` is a large integer, the largest that can be expressed without using bignums. Its value depends on the implementation, but in most Lisps it is over 16 million.

strategies the light spreads out as we search deeper, but in beam search the light remains tightly focused. Beam search is a variant of best-first search, but it is also similar to depth-first search. The difference is that beam search looks down several paths at once, instead of just one, and chooses the best one to look at next. But it gives up the ability to backtrack indefinitely. The function `beam-search` is just like `best-first-search`, except that after we sort the states, we then take only the first `beam-width` states. This is done with `subseq`; (`subseq list start end`) returns the sublist that starts at position `start` and ends just before position `end`.

```
(defun beam-search (start goal-p successors cost-fn beam-width)
  "Search highest scoring states first until goal is reached,
  but never consider more than beam-width states at a time."
  (tree-search (list start) goal-p successors
    #'(lambda (old new)
        (let ((sorted (funcall (sorter cost-fn) old new)))
          (if (> beam-width (length sorted))
              sorted
              (subseq sorted0 beam-width))))))
```

We can successfully search for 12 in the binary tree using a beam width of only 2:

```
> (beam-search 1 (is 12) #'binary-tree (price-is-right 12) 2)
;; Search: (1)
;; Search: (3 2)
;; Search: (7 6)
;; Search: (6 14)
;; Search: (12 13)
12
```

However, if we go back to the scoring function that just takes the difference from 12, then beam search fails. When it generates 14 and 15, it throws away 6, and thus loses its only chance to find the goal:

```
> (beam-search 1 (is 12) #'binary-tree (diff 12) 2)
;; Search: (1)
;; Search: (3 2)
;; Search: (7 6)
;; Search: (14 15)
;; Search: (15 28)
;; Search: (28 30)
;; Search: (30 56)
;; Search: (56 60)
;; Search: (60 112)
;; Search: (112 120)
;; Search: (120 224)
```

[Abort]

This search would succeed if we gave a beam width of 3. This illustrates a general principle: we can find a goal either by looking at more states, or by being smarter about the states we look at. That means having a better ordering function.

Notice that with a beam width of infinity we get best-first search. With a beam width of 1, we get depth-first search with no backup. This could be called “depth-only search,” but it is more commonly known as *hill-climbing*. Think of a mountaineer trying to reach a peak in a heavy fog. One strategy would be for the mountaineer to look at adjacent locations, climb to the highest one, and look again. This strategy may eventually hit the peak, but it may also get stuck at the top of a foothill, or *local maximum*. Another strategy would be for the mountaineer to turn back and try again when the fog lifts, but in AI, unfortunately, the fog rarely lifts.<sup>4</sup>

As a concrete example of a problem that can be solved by search, consider the task of planning a flight across the North American continent in a small airplane, one whose range is limited to 1000 kilometers. Suppose we have a list of selected cities with airports, along with their position in longitude and latitude:

```
(defstruct (city (:type list)) name long lat)
(defparameter *cities*
  '((Atlanta     84.23 33.45) (Los-Angeles   118.15 34.03)
    (Boston      71.05 42.21) (Memphis       90.03 35.09)
    (Chicago     87.37 41.50) (New-York      73.58 40.47)
    (Denver      105.00 39.45) (Oklahoma-City 97.28 35.26)
    (Eugene      123.05 44.03) (Pittsburgh    79.57 40.27)
    (Flagstaff   111.41 35.13) (Quebec       71.11 46.49)
    (Grand-Jct   108.37 39.05) (Reno          119.49 39.30)
    (Houston     105.00 34.00) (San-Francisco 122.26 37.47)
    (Indianapolis 86.10 39.46) (Tampa         82.27 27.57)
    (Jacksonville 81.40 30.22) (Victoria     123.21 48.25)
    (Kansas-City  94.35 39.06) (Wilmington   77.57 34.14)))
```

This example introduces a new option to `defstruct`. Instead of just giving the name of the structure, it is also possible to use:

```
(defstruct (structure-name (option value)... ) "optional doc" slot...)
```

For `city`, the option `:type` is specified as `list`. This means that `cities` will be implemented as lists of three elements, as they are in the initial value for `*cities*`.

---

<sup>4</sup>In chapter 8 we will see an example where the fog did lift: symbolic integration was once handled as a problem in search, but new mathematical results now make it possible to solve the same class of integration problems without search.

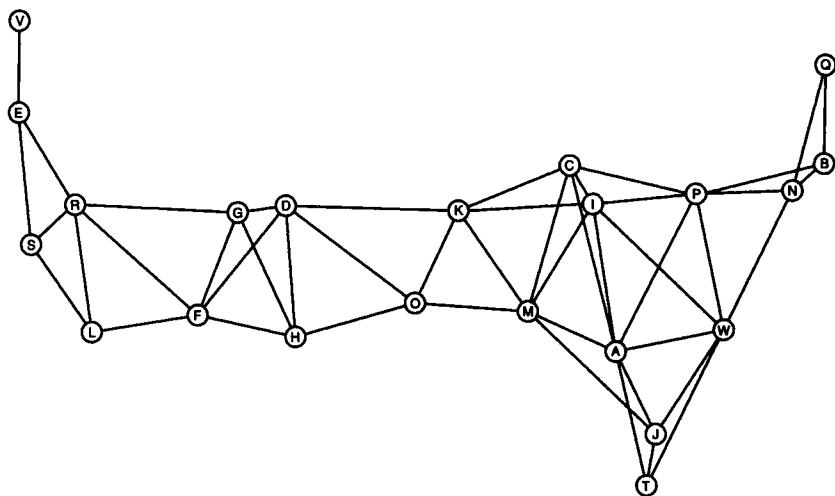


Figure 6.1: A Map of Some Cities

The cities are shown on the map in figure 6.1, which has connections between all cities within the 1000 kilometer range of each other.<sup>5</sup> This map was drawn with the help of `air-distance`, a function that returns the distance in kilometers between two cities “as the crow flies.” It will be defined later. Two other useful functions are `neighbors`, which finds all the cities within 1000 kilometers, and `city`, which maps from a name to a city. The former uses `find-all-if`, which was defined on page 101 as a synonym for `remove-if-not`.

```

(defun neighbors (city)
  "Find all cities within 1000 kilometers."
  (find-all-if #'(lambda (c)
    (and (not (eq c city))
         (< (air-distance c city) 1000.0)))
    *cities*))

(defun city (name)
  "Find the city with this name."
  (assoc name *cities*))
  
```

We are now ready to plan a trip. The function `trip` takes the name of a starting and destination city and does a beam search of width one, considering all neighbors as

---

<sup>5</sup>The astute reader will recognize that this graph is not a tree. The difference between trees and graphs and the implications for searching will be covered later.

successors to a state. The cost for a state is the air distance to the destination city:

```
(defun trip (start dest)
  "Search for a way from the start to dest."
  (beam-search start (is dest) #'neighbors
    #'(lambda (c) (air-distance c dest))
    1))
```

Here we plan a trip from San Francisco to Boston. The result seems to be the best possible path:

```
> (trip (city 'san-francisco) (city 'boston))
;; Search: ((SAN-FRANCISCO 122.26 37.47))
;; Search: ((RENO 119.49 39.3))
;; Search: ((GRAND-JCT 108.37 39.05))
;; Search: ((DENVER 105.0 39.45))
;; Search: ((KANSAS-CITY 94.35 39.06))
;; Search: ((INDIANAPOLIS 86.1 39.46))
;; Search: ((PITTSBURGH 79.57 40.27))
;; Search: ((BOSTON 71.05 42.21))
(BOSTON 71.05 42.21)
```

But look what happens when we plan the return trip. There are two detours, to Chicago and Flagstaff:

```
> (trip (city 'boston) (city 'san-francisco))
;; Search: ((BOSTON 71.05 42.21))
;; Search: ((PITTSBURGH 79.57 40.27))
;; Search: ((CHICAGO 87.37 41.5))
;; Search: ((KANSAS-CITY 94.35 39.06))
;; Search: ((DENVER 105.0 39.45))
;; Search: ((FLAGSTAFF 111.41 35.13))
;; Search: ((RENO 119.49 39.3))
;; Search: ((SAN-FRANCISCO 122.26 37.47))
(SAN-FRANCISCO 122.26 37.47)
```

Why did `trip` go from Denver to San Francisco via Flagstaff? Because Flagstaff is closer to the destination than Grand Junction. The problem is that we are minimizing the distance to the destination at each step, when we should be minimizing the sum of the distance to the destination plus the distance already traveled.

### *Search Paths*

To minimize the total distance, we need some way to talk about the *path* that leads to the goal. But the functions we have defined so far only deal with individual states along the way. Representing paths would lead to another advantage: we could return the path as the solution, rather than just return the goal state. As it is, `trip` only returns the goal state, not the path to it. So there is no way to determine what `trip` has done, except by reading the debugging output.

The data structure `path` is designed to solve both these problems. A path has four fields: the current state, the previous partial path that this path is extending, the cost of the path so far, and an estimate of the total cost to reach the goal. Here is the structure definition for `path`. It uses the `:print-function` option to say that all paths are to be printed with the function `print-path`, which will be defined below.

```
(defstruct (path (:print-function print-path))
  state (previous nil) (cost-so-far 0) (total-cost 0))
```

The next question is how to integrate paths into the searching routines with the least amount of disruption. Clearly, it would be better to make one change to `tree-search` rather than to change `depth-first-search`, `breadth-first-search`, and `beam-search`. However, looking back at the definition of `tree-search`, we see that it makes no assumptions about the structure of states, other than the fact that they can be manipulated by the goal predicate, successor, and combiner functions. This suggests that we can use `tree-search` unchanged if we pass it paths instead of states, and give it functions that can process paths.

In the following redefinition of `trip`, the `beam-search` function is called with five arguments. Instead of passing it a city as the start state, we pass a path that has the city as its state field. The goal predicate should test whether its argument is a path whose state is the destination; we assume (and later define) a version of `is` that accommodates this. The successor function is the most difficult. Instead of just generating a list of neighbors, we want to first generate the neighbors, then make each one into a path that extends the current path, but with an updated cost so far and total estimated cost. The function `path-saver` returns a function that will do just that. Finally, the cost function we are trying to minimize is `path-total-cost`, and we provide a beam width, which is now an optional argument to `trip` that defaults to one:

```
(defun trip (start dest &optional (beam-width 1))
  "Search for the best path from the start to dest."
  (beam-search
    (make-path :state start)
    (is dest :key #'path-state)
    (path-saver #'neighbors #'air-distance
```

```

  #'(lambda (c) (air-distance c dest)))
  #'path-total-cost
  beam-width))

```

The calculation of `air-distance` involves some complicated conversion of longitude and latitude to x-y-z coordinates. Since this is a problem in solid geometry, not AI, the code is presented without further comment:

```

(defconstant earth-diameter 12765.0
  "Diameter of planet earth in kilometers.")

(defun air-distance (city1 city2)
  "The great circle distance between two cities."
  (let ((d (distance (xyz-coords city1) (xyz-coords city2))))
    ; d is the straight-line chord between the two cities,
    ; The length of the subtending arc is given by:
    (* earth-diameter (asin (/ d 2)))))

(defun xyz-coords (city)
  "Returns the x,y,z coordinates of a point on a sphere.
  The center is (0 0 0) and the north pole is (0 0 1)."
  (let ((psi (deg->radians (city-lat city)))
        (phi (deg->radians (city-long city))))
    (list (* (cos psi) (cos phi))
          (* (cos psi) (sin phi))
          (sin psi)))

(defun distance (point1 point2)
  "The Euclidean distance between two points.
  The points are coordinates in n-dimensional space."
  (sqrt (reduce #'+ (mapcar #'(lambda (a b) (expt (- a b) 2))
                            point1 point2)))

(defun deg->radians (deg)
  "Convert degrees and minutes to radians."
  (* (+ (truncate deg) (* (rem deg 1) 100/60)) pi 1/180))

```

Before showing the auxiliary functions that implement this, here are some examples that show what it can do. With a beam width of 1, the detour to Flagstaff is eliminated, but the one to Chicago remains. With a beam width of 3, the correct optimal path is found. In the following examples, each call to the new version of `trip` returns a path, which is printed by `show-city-path`:

```

> (show-city-path (trip (city 'san-francisco) (city 'boston) 1))
#<Path 4514.8 km: San-Francisco - Reno - Grand-Jct - Denver -
  Kansas-City - Indianapolis - Pittsburgh - Boston>

```

```
> (show-city-path (trip (city 'boston) (city 'san-francisco) 1))
#<Path 4577.3 km: Boston - Pittsburgh - Chicago - Kansas-City -
Denver - Grand-Jct - Reno - San-Francisco>

> (show-city-path (trip (city 'boston) (city 'san-francisco) 3))
#<Path 4514.8 km: Boston - Pittsburgh - Indianapolis -
Kansas-City - Denver - Grand-Jct - Reno - San-Francisco>
```

This example shows how search is susceptible to irregularities in the search space. It was easy to find the correct path from west to east, but the return trip required more search, because Flagstaff is a falsely promising step. In general, there may be even worse dead ends lurking in the search space. Look what happens when we limit the airplane's range to 700 kilometers. The map is shown in figure 6.2.

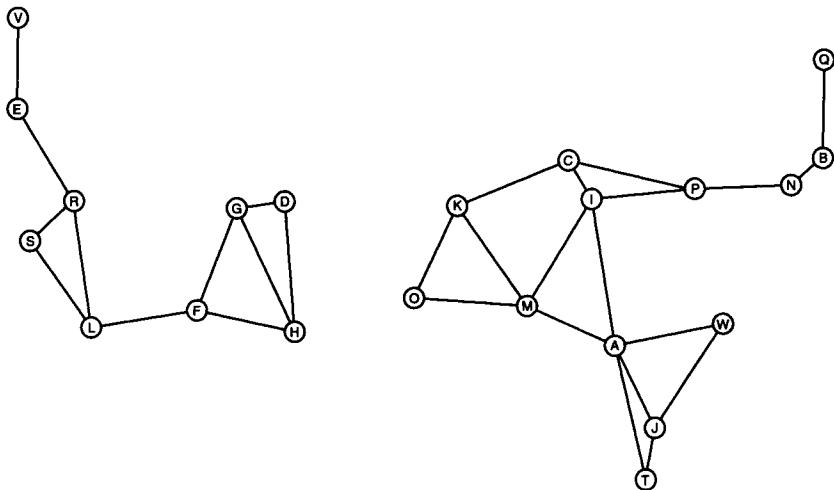


Figure 6.2: A Map of Cities within 700km

If we try to plan a trip from Tampa to Quebec, we can run into problems with the dead end at Wilmington, North Carolina. With a beam width of 1, the path to Jacksonville and then Wilmington will be tried first. From there, each step of the path alternates between Atlanta and Wilmington. The search never gets any closer to the goal. But with a beam width of 2, the path from Tampa to Atlanta is not discarded, and it is eventually continued on to Indianapolis and eventually to Quebec. So the capability to back up is essential in avoiding dead ends.

Now for the implementation details. The function is still returns a predicate that tests for a value, but now it accepts :key and :test keywords:

---

```
(defun is (value &key (key #'identity) (test #'eq))
  "Returns a predicate that tests for a given value."
  #'(lambda (path) (funcall test value (funcall key path))))
```

The `path-saver` function returns a function that will take a path as an argument and generate successors paths. `path-saver` takes as an argument a successor function that operates on bare states. It calls this function and, for each state returned, builds up a path that extends the existing path and stores the cost of the path so far as well as the estimated total cost:

```
(defun path-saver (successors cost-fn cost-left-fn)
  #'(lambda (old-path)
    (let ((old-state (path-state old-path)))
      (mapcar
        #'(lambda (new-state)
          (let ((old-cost
                  (+ (path-cost-so-far old-path)
                     (funcall cost-fn old-state new-state))))
            (make-path
              :state new-state
              :previous old-path
              :cost-so-far old-cost
              :total-cost (+ old-cost (funcall cost-left-fn
                                                new-state))))
          (funcall successors old-state)))))
```

By default a path structure would be printed as `#S(PATH ...)`. But because each path has a previous field that is filled by another path, this output would get quite verbose. That is why we installed `print-path` as the print function for paths when we defined the structure. It uses the notation `#<...>`, which is a Common Lisp convention for printing output that can not be reconstructed by `read`. The function `show-city-path` prints a more complete representation of a path. We also define `map-path` to iterate over a path, collecting values:

```
(defun print-path (path &optional (stream t) depth)
  (declare (ignore depth))
  (format stream "#<Path to ~a cost ~,1f>"
         (path-state path) (path-total-cost path)))

(defun show-city-path (path &optional (stream t))
  "Show the length of a path, and the cities along it."
  (format stream "#<Path ~,1f km: ~{~:(~a~)~~ - ~}>"
         (path-total-cost path)
         (reverse (map-path #'city-name path)))
  (values))
```

```
(defun map-path (fn path)
  "Call fn on each state in the path, collecting results."
  (if (null path)
      nil
      (cons (funcall fn (path-state path))
            (map-path fn (path-previous path)))))
```

### *Guessing versus Guaranteeing a Good Solution*

Elementary AI textbooks place a great emphasis on search algorithms that are guaranteed to find the best solution. However, in practice these algorithms are hardly ever used. The problem is that guaranteeing the best solution requires looking at a lot of other solutions in order to rule them out. For problems with large search spaces, this usually takes too much time. The alternative is to use an algorithm that will probably return a solution that is close to the best solution, but gives no guarantee. Such algorithms, traditionally known as *non-admissible heuristic search* algorithms, can be much faster.

Of the algorithms we have seen so far, best-first search almost, but not quite, guarantees the best solution. The problem is that it terminates a little too early. Suppose it has calculated three paths, of cost 90, 95 and 110. It will expand the 90 path next. Suppose this leads to a solution of total cost 100. Best-first search will then return that solution. But it is possible that the 95 path could lead to a solution with a total cost less than 100. Perhaps the 95 path is only one unit away from the goal, so it could result in a complete path of length 96. This means that an optimal search should examine the 95 path (but not the 110 path) before exiting.

Depth-first search and beam search, on the other hand, are definitely heuristic algorithms. Depth-first search finds a solution without any regard to its cost. With beam search, picking a good value for the beam width can lead to a good, quick solution, while picking the wrong value can lead to failure, or to a poor solution. One way out of this dilemma is to start with a narrow beam width, and if that does not lead to an acceptable solution, widen the beam and try again. We will call this *iterative widening*, although that is not a standard term. There are many variations on this theme, but here is a simple one:

```
(defun iter-wide-search (start goal-p successors cost-fn
                           &key (width 1) (max 100))
  "Search, increasing beam width from width to max.
  Return the first solution found at any width."
  (dbg :search "; Width: ~d" width)
  (unless (> width max)
    (or (beam-search start goal-p successors cost-fn width)
        (iter-wide-search start goal-p successors cost-fn
```

```
:width (+ width 1) :max max))))
```

Here `iter-wide-search` is used to search through a binary tree, failing with beam width 1 and 2, and eventually succeeding with beam width 3:

```
> (iter-wide-search 1 (is 12) (finite-binary-tree 15) (diff 12))
: Width: 1
;; Search: (1)
;; Search: (3)
;; Search: (7)
;; Search: (14)
;; Search: NIL
: Width: 2
;; Search: (1)
;; Search: (3 2)
;; Search: (7 6)
;; Search: (14 15)
;; Search: (15)
;; Search: NIL
: Width: 3
;; Search: (1)
;; Search: (3 2)
;; Search: (7 6 2)
;; Search: (14 15 6)
;; Search: (15 6)
;; Search: (6)
;; Search: (12 13)
12
```

The name iterative widening is derived from the established term *iterative deepening*. Iterative deepening is used to control depth-first search when we don't know the depth of the desired solution. The idea is first to limit the search to a depth of 1, then 2, and so on. That way we are guaranteed to find a solution at the minimum depth, just as in breadth-first search, but without wasting as much storage space. Of course, iterative deepening does waste some time because at each increasing depth it repeats all the work it did at the previous depth. But suppose that the average state has ten successors. That means that increasing the depth by one results in ten times more search, so only 10% of the time is wasted on repeated work. So iterative deepening uses only slightly more time and much less space. We will see it again in chapters 11 and 18.

## Searching Graphs

So far, tree-search has been the workhorse behind all the searching routines. This is curious, when we consider that the city problem involves a graph that is not a tree at all. The reason tree-search works is that any graph can be treated as a tree, if we ignore the fact that certain nodes are identical. For example, the graph in figure 6.3 can be rendered as a tree. Figure 6.4 shows only the top four levels of the tree; each of the bottom nodes (except the 6s) needs to be expanded further.

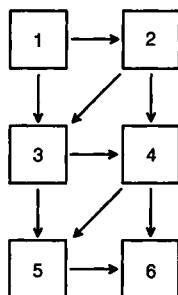


Figure 6.3: A Graph with Six Nodes

In searching for paths through the graph of cities, we were implicitly turning the graph into a tree. That is, if tree-search found two paths from Pittsburgh to Kansas City (via Chicago or Indianapolis), then it would treat them as two independent paths, just as if there were two distinct Kansas Cities. This made the algorithms simpler, but it also doubles the number of paths left to examine. If the destination is San Francisco, we will have to search for a path from Kansas City to San Francisco twice instead of once. In fact, even though the graph has only 22 cities, the tree is infinite, because we can go back and forth between adjacent cities any number of times. So, while it is possible to treat the graph as a tree, there are potential savings in treating it as a true graph.

The function `graph-search` does just that. It is similar to `tree-search`, but accepts two additional arguments: a comparison function that tests if two states are equal, and a list of states that are no longer being considered, but were examined in the past. The difference between `graph-search` and `tree-search` is in the call to `new-states`, which generates successors but eliminates states that are in either the list of states currently being considered or the list of old states considered in the past.

```
(defun graph-search (states goal-p successors combiner
                    &optional (state= #'eq) old-states)
  "Find a state that satisfies goal-p. Start with states,
```

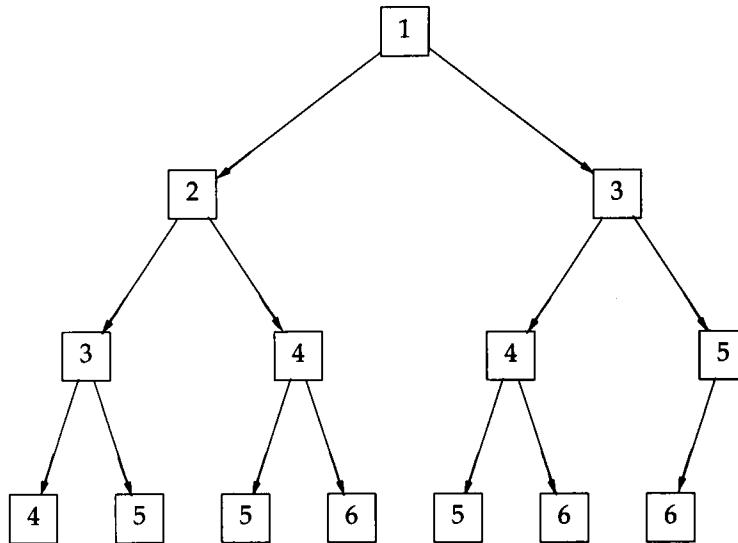


Figure 6.4: The Corresponding Tree

```

and search according to successors and combiner.
Don't try the same state twice."
(dbg :search "~&;; Search: ~a" states)
(cond ((null states) fail)
      ((funcall goal-p (first states)) (first states))
      (t (graph-search
           (funcall
            combiner
            (new-states states successors state= old-states)
            (rest states))
           goal-p successors combiner state=
           (adjoin (first states) old-states
                   :test state)))))

(defun new-states (states successors state= old-states)
  "Generate successor states that have not been seen before."
  (remove-if
   #'(lambda (state)
       (or (member state states :test state=)
           (member state old-states :test state=)))
   (funcall successors (first states))))
  
```

Using the successor function `next2`, we can search the graph shown here either as a tree or as a graph. If we search it as a graph, it takes fewer iterations and less storage space to find the goal. Of course, there is additional overhead to test for identical

states, but on graphs like this one we get an exponential speed-up for a constant amount of overhead.

```
(defun next2 (x) (list (+ x 1) (+ x 2)))

> (tree-search '(1) (is 6 #'next2 #'prepend)
  :: Search: (1)
  :: Search: (2 3)
  :: Search: (3 3 4)
  :: Search: (3 4 4 5)
  :: Search: (4 4 5 4 5)
  :: Search: (4 5 4 5 5 6)
  :: Search: (5 4 5 5 6 5 6)
  :: Search: (4 5 5 6 5 6 6 7)
  :: Search: (5 5 6 5 6 6 7 5 6)
  :: Search: (5 6 5 6 6 7 5 6 6 7)
  :: Search: (6 5 6 6 7 5 6 6 7 6 7)
  6

> (graph-search '(1) (is 6 #'next2 #'prepend)
  :: Search: (1)
  :: Search: (2 3)
  :: Search: (3 4)
  :: Search: (4 5)
  :: Search: (5 6)
  :: Search: (6 7)
  6
```

The next step is to extend the graph-search algorithm to handle paths. The complication is in deciding which path to keep when two paths reach the same state. If we have a cost function, then the answer is easy: keep the path with the cheaper cost. Best-first search of a graph removing duplicate states is called *A\* search*.

*A\** search is more complicated than graph-search because of the need both to add and to delete paths to the lists of current and old paths. For each new successor state, there are three possibilities. The new state may be in the list of current paths, in the list of old paths, or in neither. Within the first two cases, there are two subcases. If the new path is more expensive than the old one, then ignore the new path—it can not lead to a better solution. If the new path is cheaper than a corresponding path in the list of current paths, then replace it with the new path. If it is cheaper than a corresponding path in the list of the old paths, then remove that old path, and put the new path in the list of current paths.

Also, rather than sort the paths by total cost on each iteration, they are kept sorted, and new paths are inserted into the proper place one at a time using `insert-path`. Two more functions, `better-path` and `find-path`, are used to compare paths and see if a state has already appeared.

```
(defun a*-search (paths goal-p successors cost-fn cost-left-fn
  &optional (state= #'eql) old-paths)
  "Find a path whose state satisfies goal-p. Start with paths,
  and expand successors, exploring least cost first.
  When there are duplicate states, keep the one with the
  lower cost and discard the other."
  (dbg :search ";; Search: ~a" paths)
  (cond
    ((null paths) fail)
    ((funcall goal-p (path-state (first paths)))
     (values (first paths) paths))
    (t (let* ((path (pop paths))
              (state (path-state path)))
        ;; Update PATHS and OLD-PATHS to reflect
        ;; the new successors of STATE:
        (setf old-paths (insert-path path old-paths))
        (dolist (state2 (funcall successors state))
          (let* ((cost (+ (path-cost-so-far path)
                           (funcall cost-fn state state2)))
                 (cost2 (funcall cost-left-fn state2))
                 (path2 (make-path
                          :state state2 :previous path
                          :cost-so-far cost
                          :total-cost (+ cost cost2)))
                 (old nil)
                 ;; Place the new path, path2, in the right list:
                 (cond
                   ((setf old (find-path state2 paths state=))
                    (when (better-path path2 old)
                      (setf paths (insert-path
                                    path2 (delete old paths))))))
                   ((setf old (find-path state2 old-paths state=))
                    (when (better-path path2 old)
                      (setf paths (insert-path path2 paths))
                      (setf old-paths (delete old old-paths)))))
                   (t (setf paths (insert-path path2 paths)))))))
        ;; Finally, call A* again with the updated path lists:
        (a*-search paths goal-p successors cost-fn cost-left-fn
                   state= old-paths))))
```

Here are the three auxiliary functions:

```
(defun find-path (state paths state=)
  "Find the path with this state among a list of paths."
  (find state paths :key #'path-state :test state=))

(defun better-path (path1 path2)
  "Is path1 cheaper than path2?"
  (< (path-total-cost path1) (path-total-cost path2)))

(defun insert-path (path paths)
  "Put path into the right position, sorted by total cost."
  ;; MERGE is a built-in function
  (merge 'list (list path) paths #'< :key #'path-total-cost))

(defun path-states (path)
  "Collect the states along this path."
  (if (null path)
      nil
      (cons (path-state path)
            (path-states (path-previous path))))))
```

Below we use a\*-search to search for 6 in the graph previously shown in figure 6.3. The cost function is a constant 1 for each step. In other words, the total cost is the length of the path. The heuristic evaluation function is just the difference from the goal. The A\* algorithm needs just three search steps to come up with the optimal solution. Contrast that to the graph search algorithm, which needed five steps, and the tree search algorithm, which needed ten steps—and neither of them found the optimal solution.

```
> (path-states
  (a*-search (list (make-path :state 1)) (is 6)
              #'next2 #'(lambda (x y) 1) (diff 6)))
;; Search: (#<Path to 1 cost 0.0>)
;; Search: (#<Path to 3 cost 4.0> #<Path to 2 cost 5.0>)
;; Search: (#<Path to 5 cost 3.0> #<Path to 4 cost 4.0>
;;           #<Path to 2 cost 5.0>)
;; Search: (#<Path to 6 cost 3.0> #<Path to 7 cost 4.0>
;;           #<Path to 4 cost 4.0> #<Path to 2 cost 5.0>)
(6 5 3 1)
```

It may seem limiting that these search functions all return a single answer. In some applications, we may want to look at several solutions, or at all possible solutions. Other applications are more naturally seen as optimization problems, where we don't know ahead of time what counts as achieving the goal but are just trying to find some action with a low cost.

It turns out that the functions we have defined are not limiting at all in this respect. They can be used to serve both these new purposes—provided we carefully specify the goal predicate. To find all solutions to a problem, all we have to do is pass in a goal predicate that always fails, but saves all the solutions in a list. The goal predicate will see all possible solutions and save away just the ones that are real solutions. Of course, if the search space is infinite this will never terminate, so the user has to be careful in applying this technique. It would also be possible to write a goal predicate that stopped the search after finding a certain number of solutions, or after looking at a certain number of states. Here is a function that finds all solutions, using beam search:

```
(defun search-all (start goal-p successors cost-fn beam-width)
  "Find all solutions to a search problem, using beam search."
  ;;= Be careful: this can lead to an infinite loop.
  (let ((solutions nil))
    (beam-search
      start #'(lambda (x)
        (when (funcall goal-p x) (push x solutions)))
      nil)
    successors cost-fn beam-width)
  solutions))
```

## 6.5 GPS as Search

The GPS program can be seen as a problem in search. For example, in the three-block blocks world, there are only 13 different states. They could be arranged in a graph and searched just as we searched for a route between cities. Figure 6.5 shows this graph.

The function `search-gps` does just that. Like the `gps` function on page 135, it computes a final state and then picks out the actions that lead to that state. But it computes the state with a beam search. The goal predicate tests if the current state satisfies every condition in the goal, the successor function finds all applicable operators and applies them, and the cost function simply sums the number of actions taken so far, plus the number of conditions that are not yet satisfied:

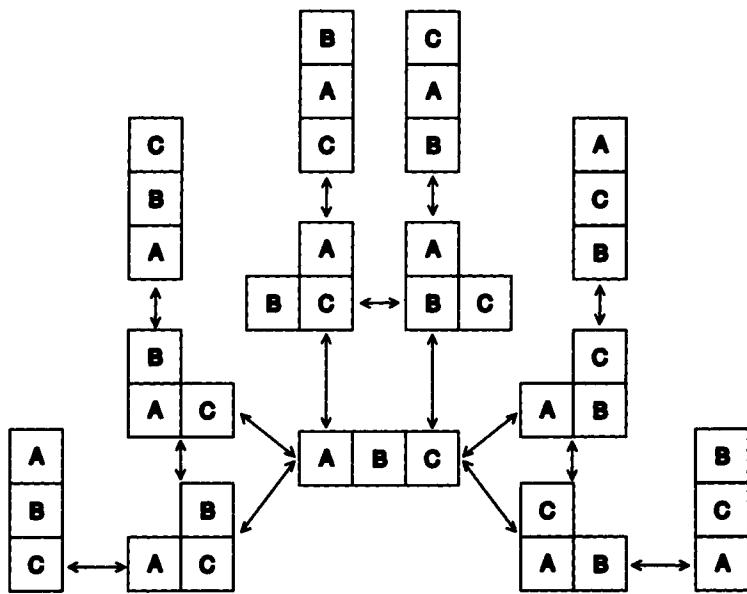


Figure 6.5: The Blocks World as a Graph

```
(defun search-gps (start goal &optional (beam-width 10))
  "Search for a sequence of operators leading to goal."
  (find-all-if
   #'action-p
   (beam-search
    (cons '(start) start)
    #'(lambda (state) (subsetp goal state :test #'equal))
    #'gps-successors
    #'(lambda (state)
        (+ (count-if #'action-p state)
           (count-if #'(lambda (con)
                         (not (member-equal con state)))
                      goal)))
    beam-width)))
```

Here is the successor function:

```
(defun gps-successors (state)
  "Return a list of states reachable from this one using ops."
  (mapcar
   #'(lambda (op)
```

---

```

  (append
    (remove-if #'(lambda (x)
                  (member-equal x (op-del-list op)))
                state)
    (op-add-list op)))
  (applicable-ops state)))

(defun applicable-ops (state)
  "Return a list of all ops that are applicable now."
  (find-all-if
    #'(lambda (op)
        (subsetp (op-preconds op) state :test #'equal))
    *ops*))

```

The search technique finds good solutions quickly for a variety of problems. Here we see the solution to the Sussman anomaly in the three-block blocks world:

```

(setf start '((c on a) (a on table) (b on table) (space on c)
              (space on b) (space on table)))

> (search-gps start '((a on b) (b on c)))
((START)
 (EXECUTING (MOVE C FROM A TO TABLE))
 (EXECUTING (MOVE B FROM TABLE TO C))
 (EXECUTING (MOVE A FROM TABLE TO B)))

> (search-gps start '((b on c) (a on b)))
((START)
 (EXECUTING (MOVE C FROM A TO TABLE))
 (EXECUTING (MOVE B FROM TABLE TO C))
 (EXECUTING (MOVE A FROM TABLE TO B)))

```

In these solutions we search forward from the start to the goal; this is quite different from the means-ends approach of searching backward from the goal for an appropriate operator. But we could formulate means-ends analysis as forward search simply by reversing start and goal: GPS's goal state is the search's start state, and the search's goal predicate tests to see if a state matches GPS's start state. This is left as an exercise.

## 6.6 History and References

Pattern matching is one of the most important tools for AI. As such, it is covered in most textbooks on Lisp. Good treatments include Abelson and Sussman (1984), Wilensky (1986), Winston and Horn (1988), and Kreutzer and McKenzie (1990). An overview is presented in the “pattern-matching” entry in *Encyclopedia of AI* (Shapiro 1990).

Nilsson's *Problem-Solving Methods in Artificial Intelligence* (1971) was an early textbook that emphasized search as the most important defining characteristic of AI. More recent texts give less importance to search; Winston's *Artificial Intelligence* (1984) gives a balanced overview, and his *Lisp* (1988) provides implementations of some of the algorithms. They are at a lower level of abstraction than the ones in this chapter. Iterative deepening was first presented by Korf (1985), and iterative broadening by Ginsberg and Harvey (1990).

## 6.7 Exercises

-  **Exercise 6.3 [m]** Write a version of `interactive-interpreter` that is more general than the one defined in this chapter. Decide what features can be specified, and provide defaults for them.
-  **Exercise 6.4 [m]** Define a version of `compose` that allows any number of arguments, not just two. Hint: You may want to use the function `reduce`.
-  **Exercise 6.5 [m]** Define a version of `compose` that allows any number of arguments but is more efficient than the answer to the previous exercise. Hint: try to make decisions when `compose` is called to build the resulting function, rather than making the same decisions over and over each time the resulting function is called.
-  **Exercise 6.6 [m]** One problem with `pat-match` is that it gives special significance to symbols starting with ?, which means that they can not be used to match a literal pattern. Define a pattern that matches the input literally, so that such symbols can be matched.
-  **Exercise 6.7 [m]** Discuss the pros and cons of data-driven programming compared to the conventional approach.
-  **Exercise 6.8 [m]** Write a version of `tree-search` using an explicit loop rather than recursion.
-  **Exercise 6.9 [m]** The `sorter` function is inefficient for two reasons: it calls `append`, which has to make a copy of the first argument, and it sorts the entire result, rather than just inserting the new states into the already sorted old states. Write a more efficient `sorter`.

- ❑ **Exercise 6.10 [m]** Write versions of graph-search and a\*-search that use hash tables rather than lists to test whether a state has been seen before.
- ❑ **Exercise 6.11 [m]** Write a function that calls beam-search to find the first  $n$  solutions to a problem and returns them in a list.
- ❑ **Exercise 6.12 [m]** On personal computers without floating-point hardware, the air-distance calculation will be rather slow. If this is a problem for you, arrange to compute the xyz-coords of each city only once and then store them, or store a complete table of air distances between cities. Also precompute and store the neighbors of each city.
- ❑ **Exercise 6.13 [d]** Write a version of GPS that uses A\* search instead of beam search. Compare the two versions in a variety of domains.
- ❑ **Exercise 6.14 [d]** Write a version of GPS that allows costs for each operator. For example, driving the child to school might have a cost of 2, but calling a limousine to transport the child might have a cost of 100. Use these costs instead of a constant cost of 1 for each operation.
- ❑ **Exercise 6.15 [d]** Write a version of GPS that uses the searching tools but does means-ends analysis.

## 6.8 Answers

**Answer 6.2** Unfortunately, pat-match does not always find the answer. The problem is that it will only rebind a segment variable based on a failure to match the rest of the pattern after the segment variable. In all the examples above, the “rest of the pattern after the segment variable” was the whole pattern, so pat-match always worked properly. But if a segment variable appears nested inside a list, then the rest of the segment variable’s sublist is only a part of the rest of the whole pattern, as the following example shows:

```
> (pat-match '(((?* ?x) (?* ?y)) ?x ?y)  
  '(( a b c d ) ( a b ) ( c d ))) ⇒ NIL
```

The correct answer with ?x bound to (a b) and ?y bound to (c d) is not found because the inner segment match succeeds with ?x bound to () and ?y bound to (a

`b c d`), and once we leave the inner match and return to the top level, there is no going back for alternative bindings.

**Answer 6.3** The following version lets the user specify all four components of the prompt-read-eval-print loop, as well as the streams to use for input and output. Defaults are set up as for a Lisp interpreter.

```
(defun interactive-interpreter
  (&key (read #'read) (eval #'eval) (print #'print)
        (prompt "> ") (input t) (output t))
  "Read an expression, evaluate it, and print the result."
  (loop
    (fresh-line output)
    (princ prompt output)
    (funcall print (funcall eval (funcall read input)))
    output)))
```

Here is another version that does all of the above and also handles multiple values and binds the various “history variables” that the Lisp top-level binds.

```
(defun interactive-interpreter
  (&key (read #'read) (eval #'eval) (print #'print)
        (prompt "> ") (input t) (output t))
  "Read an expression, evaluate it, and print the result(s).
  Does multiple values and binds: * *** *** - + ++ +++ / // ///"
  (let (* *** *** - + ++ +++ / // // vals)
    ;: The above variables are all special, except VALS
    ;: The variable - holds the current input
    ;: * *** *** are the 3 most recent values
    ;: + ++ +++ are the 3 most recent inputs
    ;: / // // are the 3 most recent lists of multiple-values
    (loop
      (fresh-line output)
      (princ prompt output)
      ;: First read and evaluate an expression
      (setf - (funcall read input)
            vals (multiple-value-list (funcall eval -)))
      ;: Now update the history variables
      (setf +++ ++     // /           *** (first //)
            ++ +       // /           ** (first //)
            + -       /   vals      *   (first //))
      ;: Finally print the computed value(s)
      (dolist (value vals)
        (funcall print value output)))))
```

**Answer 6.4**

```
(defun compose (&rest functions)
  "Return the function that is the composition of all the args.
  i.e. (compose f g h) = (lambda (x) (f (g (h x))))."
  #'(lambda (x)
      (reduce #'funcall functions :from-end t :initial-value x)))
```

**Answer 6.5**

```
(defun compose (&rest functions)
  "Return the function that is the composition of all the args.
  i.e. (compose f g h) = (lambda (x) (f (g (h x))))."
  (case (length functions)
    (0 #'identity)
    (1 (first functions))
    (2 (let ((f (first functions))
             (g (second functions)))
        #'(lambda (x) (funcall f (funcall g x)))))
    (t #'(lambda (x)
            (reduce #'funcall functions :from-end t
                   :initial-value x)))))
```

**Answer 6.8**

```
(defun tree-search (states goal-p successors combiner)
  "Find a state that satisfies goal-p. Start with states,
  and search according to successors and combiner."
  (loop
    (cond ((null states) (RETURN fail))
          ((funcall goal-p (first states))
           (RETURN (first states)))
          (t (setf states
                    (funcall combiner
                            (funcall successors (first states))
                            (rest states)))))))
```

**Answer 6.9**

```
(defun sorter (cost-fn)
  "Return a combiner function that sorts according to cost-fn."
  #'(lambda (new old)
      (merge 'list (sort new #'> :key cost-fn)
             old #'> :key cost-fn)))
```

**Answer 6.11**

```
(defun search-n (start n goal-p successors cost-fn beam-width)
  "Find n solutions to a search problem, using beam search."
  (let ((solutions nil))
    (beam-search
      start #'(lambda (x)
        (cond ((not (funcall goal-p x)) nil)
              ((= n 0) x)
              (t (decf n)
                  (push x solutions)
                  nil)))
      successors cost-fn beam-width)
    solutions))
```

## CHAPTER 7

---

# STUDENT: Solving Algebra Word Problems

*[This] is an example par excellence of the power of using meaning to solve linguistic problems.*

—Marvin Minsky (1968)  
MIT computer scientist

**S**TUDENT was another early language understanding program, written by Daniel Bobrow as his Ph.D. research project in 1964. It was designed to read and solve the kind of word problems found in high school algebra books. An example is:

*If the number of customers Tom gets is twice the square of 20% of the number of advertisements he runs, and the number of advertisements is 45, then what is the number of customers Tom gets?*

STUDENT could correctly reply that the number of customers is 162. To do this, STUDENT must be far more sophisticated than ELIZA; it must process and “understand” a great deal of the input, rather than just concentrate on a few key words. And it must compute a response, rather than just fill in blanks. However, we shall see that the STUDENT program uses little more than the pattern-matching techniques of ELIZA to translate the input into a set of algebraic equations. From there, it must know enough algebra to solve the equations, but that is not very difficult.

The version of STUDENT we develop here is nearly a full implementation of the original. However, remember that while the original was state-of-the-art as of 1964, AI has made some progress in a quarter century, as subsequent chapters will attempt to show.

## 7.1 Translating English into Equations

The description of STUDENT is:

1. Break the input into phrases that will represent equations.
2. Break each phrase into a pair of phrases on either side of the = sign.
3. Break these phrases down further into sums and products, and so on, until finally we bottom out with numbers and variables. (By “variable” here, I mean “mathematical variable,” which is distinct from the idea of a “pattern-matching variable” as used in pat-match in chapter 6).
4. Translate each English phrase into a mathematical expression. We use the idea of a rule-based translator as developed for ELIZA.
5. Solve the resulting mathematical equations, coming up with a value for each unknown variable.
6. Print the values of all the variables.

For example, we might have a pattern of the form (If ?x then ?y), with an associated response that says that ?x and ?y will each be equations or lists of equations. Applying the pattern to the input above, ?y would have the value (what is the number of customers Tom gets). Another pattern of the form (?x is ?y) could have a response corresponding to an equation where ?x and ?y are the two sides of the equation. We could then make up a mathematical variable for (what) and another for (the number of customers Tom gets). We would recognize this later phrase as a variable because there are no patterns to break it down further. In contrast, the phrase (twice the square of 20 per cent of the number of advertisements he runs) could match a pattern of the form (twice ?x) and transform to (\* 2 (the square of 20 per cent of the number of advertisements he runs)), and by further applying patterns of the form (the square of ?x) and (?x per cent of ?y) we could arrive at a final response of (\* 2 (expt (\* (/ 20 100) n) 2)), where n is the variable generated by (the number of advertisements he runs).

Thus, we need to represent variables, expressions, equations, and sets of equations. The easiest thing to do is to use something we know: represent them just as Lisp itself does. Variables will be symbols, expressions and equations will be nested

lists with prefix operators, and sets of equations will be lists of equations. With that in mind, we can define a list of pattern-response rules corresponding to the type of statements found in algebra word problems. The structure definition for a rule is repeated here, and the structure `exp`, an expression, is added. `lhs` and `rhs` stand for left- and right-hand side, respectively. Note that the constructor `mkexp` is defined as a constructor that builds expressions without taking keyword arguments. In general, the notation `(:constructor fn args)` creates a constructor function with the given name and argument list.<sup>1</sup>

```
(defstruct (rule (:type list)) pattern response)
(defstruct (exp (:type list)
               (:constructor mkexp (lhs op rhs)))
          op lhs rhs)
(defun exp-p (x) (consp x))
(defun exp-args (x) (rest x))
```

We ignored commas and periods in ELIZA, but they are crucial for STUDENT, so we must make allowances for them. The problem is that a `" , "` in Lisp normally can be used only within a backquote construction, and a `". ."` normally can be used only as a decimal point or in a dotted pair. The special meaning of these characters to the Lisp reader can be escaped either by preceding the character with a backslash (`\,.`) or by surrounding the character by vertical bars (`| , |`).

```
(pat-match-abbrev '?x* '(* ?x))
(pat-match-abbrev '?y* '(* ?y))

(defparameter *student-rules* (mapcar #'expand-pat-match-abbrev
  '(((?x* |.|) ?x)
    ((?x* |.| ?y*) (?x ?y))
    ((if ?x* |.| then ?y*) (?x ?y))
    ((if ?x* then ?y*) (?x ?y))
    ((if ?x* |.| ?y*) (?x ?y))
    ((?x* |.| and ?y*) (?x ?y))
    ((find ?x* and ?y*) ((= to-find-1 ?x) (= to-find-2 ?y)))
    ((find ?x*) (= to-find ?x))
    ((?* equals ?y*) (= ?x ?y))
    ((?* same as ?y*) (= ?x ?y))
    ((?x* = ?y*) (= ?x ?y))
    ((?x* is equal to ?y*) (= ?x ?y))
    ((?x* is ?y*) (= ?x ?y))
    ((?x* - ?y*) (- ?x ?y))
    ((?x* minus ?y*) (- ?x ?y)))
```

---

<sup>1</sup>Page 316 of *Common Lisp the Language* says, “Because a constructor of this type operates By Order of Arguments, it is sometimes known as a BOA constructor.”

```
((difference between ?x* and ?y*) (- ?y ?x))
((difference ?x* and ?y*)      (- ?y ?x))
((?x* + ?y*)      (+ ?x ?y))
((?x* plus ?y*)      (+ ?x ?y))
((sum ?x* and ?y*)      (+ ?x ?y))
((product ?x* and ?y*)      (* ?x ?y))
((?x* * ?y*)      (* ?x ?y))
((?x* times ?y*)      (* ?x ?y))
((?x* / ?y*)      (/ ?x ?y))
((?x* per ?y*)      (/ ?x ?y))
((?x* divided by ?y*)      (/ ?x ?y))
((half ?x*)      (/ ?x 2))
((one half ?x*)      (/ ?x 2))
((twice ?x*)      (* 2 ?x))
((square ?x*)      (* ?x ?x))
((?x* % less than ?y*)      (* ?y (/ (- 100 ?x) 100)))
((?x* % more than ?y*)      (* ?y (/ (+ 100 ?x) 100)))
((?x* % ?y*)      (* (/ ?x 100) ?y))))
```

The main section of STUDENT will search through the list of rules for a response, just as ELIZA did. The first point of deviation is that before we substitute the values of the pat-match variables into the response, we must first recursively translate the value of each variable, using the same list of pattern-response rules. The other difference is that once we're done, we don't just print the response; instead we have to solve the set of equations and print the answers. The program is summarized in figure 7.1.

Before looking carefully at the program, let's try a sample problem: "If z is 3, what is twice z?" Applying the rules to the input gives the following trace:

```
Input: (If z is 3, what is twice z)
Rule: ((if ?x ,! ?y)      (?x ?y))
Binding: ((?x . (z is 3))  (?y . (what is twice z)))
Input: (z is 3)
Rule: ((?x is ?y)      (= ?x ?y))
Result: (= z 3)

Input: (what is twice z ?)
Rule: ((?x is ?y)      (= ?x ?y))
Binding: ((?x . what) (?y . (twice z)))
Input: (twice z)
Rule: ((twice ?x)      (* 2 ?x))
Result: (* 2 z)
Result: (= what (* 2 z))
Result: ((= z 3) (= what (* 2 z)))
```

There are two minor complications. First, we agreed to implement sets of equations as lists of equations. For this example, everything worked out, and the response

	<b>Top-Level Function</b>
student	Solve certain algebra word problems.
*student-rules*	<b>Special Variables</b> A list of pattern/response pairs.
	<b>Data Types</b>
exp	An operator and its arguments.
rule	A pattern and response.
	<b>Major Functions</b>
translate-to-expression	Translate an English phrase into an equation or expression.
translate-pair	Translate the value part of the pair into an equation or expression.
create-list-of-equations	Separate out equations embedded in nested parens.
solve-equations	Print the equations and their solution.
solve	Solve a system of equations by constraint propagation.
	<b>Auxiliary Functions</b>
isolate	Isolate the lone variable on the left-hand side of an expression.
noise-word-p	Is this a low-content word that can be safely ignored?
make-variable	Create a variable name based on the given list of words.
print-equations	Print a list of equations.
inverse-op	I.e., the inverse of + is -.
unknown-p	Is the argument an unknown (variable)?
in-exp	True if x appears anywhere in exp.
no-unknown	Returns true if there are no unknowns in exp.
one-unknown	Returns the single unknown in exp, if there is exactly one.
commutative-p	Is the operator commutative?
solve-arithmetic	Perform arithmetic on rhs of an equation.
binary-exp-p	Is this a binary expression?
prefix->infix	Translate prefix to infix expressions.
mkexp	Make an expression.
	<b>Previously Defined Functions</b>
pat-match	Match pattern against an input. (p. 180)
rule-based-translator	Apply a set of rules. (p. 189)

Figure 7.1: Glossary for the STUDENT Program

was a list of two equations. But if nested patterns are used, the response could be something like `((= a 5) ((= b (+ a 1)) (= c (+ a b))))`, which is not a list of equations. The function `create-list-of-equations` transforms a response like this into a proper list of equations. The other complication is choosing variable names. Given a list of words like `(the number of customers Tom gets)`, we want to choose a symbol to represent it. We will see below that the symbol `customers` is chosen, but that there are other possibilities.

Here is the main function for STUDENT. It first removes words that have no content, then translates the input to one big expression with `translate-to-expression`, and breaks that into separate equations with `create-list-of-equations`. Finally, the function `solve-equations` does the mathematics and prints the solution.

```
(defun student (words)
  "Solve certain Algebra Word Problems."
  (solve-equations
    (create-list-of-equations
      (translate-to-expression (remove-if #'noise-word-p words)))))
```

The function `translate-to-expression` is a rule-based translator. It either finds some rule in `*student-rules*` to transform the input, or it assumes that the entire input represents a single variable. The function `translate-pair` takes a variable/value binding pair and translates the value by a recursive call to `translate-to-expression`.

```
(defun translate-to-expression (words)
  "Translate an English phrase into an equation or expression."
  (or (rule-based-translator
        words *student-rules*
        :rule-if #'rule-pattern :rule-then #'rule-response
        :action #'(lambda (bindings response)
                    (sublis (mapcar #'translate-pair bindings)
                           response)))
        (make-variable words)))

(defun translate-pair (pair)
  "Translate the value part of the pair into an equation or expression."
  (cons (binding-var pair)
        (translate-to-expression (binding-val pair))))
```

The function `create-list-of-equations` takes a single expression containing embedded equations and separates them into a list of equations:

```
(defun create-list-of-equations (exp)
  "Separate out equations embedded in nested parens."
  (cond ((null exp) nil)
        ((atom (first exp)) (list exp))
        (t (append (create-list-of-equations (first exp))
                   (create-list-of-equations (rest exp))))))
```

Finally, the function `make-variable` creates a variable to represent a list of words. We do that by first removing all “noise words” from the input, and then taking the first symbol that remains. So, for example, “the distance John traveled” and “the distance traveled by John” will both be represented by the same variable, `distance`, which is certainly the right thing to do. However, “the distance Mary traveled” will also be represented by the same variable, which is certainly a mistake. For (the number of customers Tom gets), the variable will be `customers`, since `the`, `of` and `number` are all noise words. This will match (`the customers` mentioned above) and

(the number of customers), but not (Tom's customers). For now, we will accept the first-non-noise-word solution, but note that exercise 7.3 asks for a correction.

```
(defun make-variable (words)
  "Create a variable name based on the given list of words"
  ;;= The list of words will already have noise words removed
  (first words))

(defun noise-word-p (word)
  "Is this a low-content word that can be safely ignored?"
  (member word '(a an the this number of $)))
```

## 7.2 Solving Algebraic Equations

The next step is to write the equation-solving section of STUDENT. This is more an exercise in elementary algebra than in AI, but it is a good example of a symbol-manipulation task, and thus an interesting programming problem.

The STUDENT program mentioned the function `solve-equations`, passing it one argument, a list of equations to be solved. `solve-equations` prints the list of equations, attempts to solve them using `solve`, and prints the result.

```
(defun solve-equations (equations)
  "Print the equations and their solution"
  (print-equations "The equations to be solved are:" equations)
  (print-equations "The solution is:" (solve equations nil)))
```

The real work is done by `solve`, which has the following specification: (1) Find an equation with exactly one occurrence of an unknown in it. (2) Transform that equation so that the unknown is isolated on the left-hand side. This can be done if we limit the operators to `+`, `-`, `*`, and `/`. (3) Evaluate the arithmetic on the right-hand side, yielding a numeric value for the unknown. (4) Substitute the numeric value for the unknown in all the other equations, and remember the known value. Then try to solve the resulting set of equations. (5) If step (1) fails—if there is no equation with exactly one unknown—then just return the known values and don't try to solve anything else.

The function `solve` is passed a system of equations, along with a list of known variable/value pairs. Initially no variables are known, so this list will be empty. `solve` goes through the list of equations searching for an equation with exactly one unknown. If it can find such an equation, it calls `isolate` to solve the equation in terms of that one unknown. `solve` then substitutes the value for the variable throughout the list of equations and calls itself recursively on the resulting list. Each

time `solve` calls itself, it removes one equation from the list of equations to be solved, and adds one to the list of known variable/value pairs. Since the list of equations is always growing shorter, `solve` must eventually terminate.

```
(defun solve (equations known)
  "Solve a system of equations by constraint propagation."
  ;;= Try to solve for one equation, and substitute its value into
  ;;= the others. If that doesn't work, return what is known.
  (or (some #'(lambda (equation)
    (let ((x (one-unknown equation)))
      (when x
        (let ((answer (solve-arithmetic
                      (isolate equation x))))
          (solve (subst (exp-rhs answer) (exp-lhs answer)
                        (remove equation equations))
                  (cons answer known))))))
      equations)
    known)))
```

`isolate` is passed an equation guaranteed to have one unknown. It returns an equivalent equation with the unknown isolated on the left-hand side. There are five cases to consider: when the unknown is alone on the left, we're done. The second case is when the unknown is anywhere on the right-hand side. Because '=' is commutative, we can reduce the problem to solving the equivalent equation with left- and right-hand sides reversed.

Next we have to deal with the case where the unknown is in a complex expression on the left-hand side. Because we are allowing four operators and the unknown can be either on the right or the left, there are eight possibilities. Letting  $X$  stand for an expression containing the unknown and  $A$  and  $B$  stand for expressions with no unknowns, the possibilities and their solutions are as follows:

(1) $X \cdot A = B \Rightarrow X = B/A$	(5) $A \cdot X = B \Rightarrow X = B/A$
(2) $X + A = B \Rightarrow X = B - A$	(6) $A + X = B \Rightarrow X = B - A$
(3) $X / A = B \Rightarrow X = B \cdot A$	(7) $A / X = B \Rightarrow X = A/B$
(4) $X - A = B \Rightarrow X = B + A$	(8) $A - X = B \Rightarrow X = A - B$

Possibilities (1) through (4) are handled by case III, (5) and (6) by case IV, and (7) and (8) by case V. In each case, the transformation does not give us the final answer, since  $X$  need not be the unknown; it might be a complex expression involving the unknown. So we have to call `isolate` again on the resulting equation. The reader should try to verify that transformations (1) to (8) are valid, and that cases III to V implement them properly.

```
(defun isolate (e x)
  "Isolate the lone x in e on the left-hand side of e."
  ;; This assumes there is exactly one x in e,
  ;; and that e is an equation.
  (cond ((eq (exp-lhs e) x)
         ;; Case I: X = A -> X = n
         e)
        ((in-exp x (exp-rhs e))
         ;; Case II: A = f(X) -> f(X) = A
         (isolate (mkexp (exp-rhs e) '= (exp-lhs e)) x))
        ((in-exp x (exp-lhs (exp-lhs e)))
         ;; Case III: f(X)*A = B -> f(X) = B/A
         (isolate (mkexp (exp-lhs (exp-lhs e)) =
                         (mkexp (exp-rhs e)
                                (inverse-op (exp-op (exp-lhs e)))
                                (exp-rhs (exp-lhs e)))) x))
        ((commutative-p (exp-op (exp-lhs e)))
         ;; Case IV: A*f(X) = B -> f(X) = B/A
         (isolate (mkexp (exp-rhs (exp-lhs e)) =
                         (mkexp (exp-rhs e)
                                (inverse-op (exp-op (exp-lhs e)))
                                (exp-lhs (exp-lhs e)))) x))
        (t ;;; Case V: A/f(X) = B -> f(X) = A/B
         (isolate (mkexp (exp-rhs (exp-lhs e)) =
                         (mkexp (exp-lhs (exp-lhs e))
                                (exp-op (exp-lhs e))
                                (exp-rhs e)))) x))))
```

Recall that to prove a function is correct, we have to prove both that it gives the correct answer when it terminates and that it will eventually terminate. For a recursive function with several alternative cases, we must show that each alternative is valid, and also that each alternative gets closer to the end in some way (that any recursive calls involve ‘simpler’ arguments). For `isolate`, elementary algebra will show that each step is valid—or at least *nearly* valid. Dividing both sides of an equation by 0 does not yield an equivalent equation, and we never checked for that. It’s also possible that similar errors could sneak in during the call to `eval`. However, if we assume the equation does have a single valid solution, then `isolate` performs only legal transformations.

The hard part is to prove that `isolate` terminates. Case I clearly terminates, and the others all contribute towards isolating the unknown on the left-hand side. For any equation, the sequence will be first a possible use of case II, followed by a number of recursive calls using cases III to V. The number of calls is bounded by the number of subexpressions in the equation, since each successive call effectively removes an expression from the left and places it on the right. Therefore, assuming the input is

of finite size, we must eventually reach a recursive call to `isolate` that will use case I and terminate.

When `isolate` returns, the right-hand side must consist only of numbers and operators. We could easily write a function to evaluate such an expression. However, we don't have to go to that effort, since the function already exists. The data structure `exp` was carefully selected to be the same structure (lists with prefix functions) used by Lisp itself for its own expressions. So Lisp will find the right-hand side to be an acceptable expression, one that could be evaluated if typed in to the top level. Lisp evaluates expressions by calling the function `eval`, so we can call `eval` directly and have it return a number. The function `solve-arithmetc` returns an equation of the form ( $= var\ number$ ).

Auxiliary functions for `solve` are shown below. Most are straightforward, but I will remark on a few of them. The function `prefix->infix` takes an expression in prefix notation and converts it to a fully parenthesized infix expression. Unlike `isolate`, it assumes the expressions will be implemented as lists. `prefix->infix` is used by `print-equations` to produce more readable output.

```
(defun print-equations (header equations)
  "Print a list of equations."
  (format t "~%~a~{~% ~{ ~a~}~}~%" header
          (mapcar #'prefix->infix equations)))

(defconstant operators-and-inverses
  '(((+ -) (- +) (* /) (/ *) (= =)))

(defun inverse-op (op)
  (second (assoc op operators-and-inverses)))

(defun unknown-p (exp)
  (symbolp exp))

(defun in-exp (x exp)
  "True if x appears anywhere in exp"
  (or (eq x exp)
      (and (exp-p exp)
           (or (in-exp x (exp-lhs exp)) (in-exp x (exp-rhs exp))))))

(defun no-unknown (exp)
  "Returns true if there are no unknowns in exp."
  (cond ((unknown-p exp) nil)
        ((atom exp) t)
        ((no-unknown (exp-lhs exp)) (no-unknown (exp-rhs exp)))
        (t nil)))
```

```

(defun one-unknown (exp)
  "Returns the single unknown in exp, if there is exactly one."
  (cond ((unknown-p exp) exp)
        ((atom exp) nil)
        ((no-unknown (exp-lhs exp)) (one-unknown (exp-rhs exp)))
        ((no-unknown (exp-rhs exp)) (one-unknown (exp-lhs exp)))
        (t nil)))

(defun commutative-p (op)
  "Is operator commutative?"
  (member op '(+ * =)))

(defun solve-arithmetic (equation)
  "Do the arithmetic for the right-hand side."
  ;;= This assumes that the right-hand side is in the right form.
  (mkexp (exp-lhs equation) '= (eval (exp-rhs equation)))))

(defun binary-exp-p (x)
  (and (exp-p x) (= (length (exp-args x)) 2)))

(defun prefix->infix (exp)
  "Translate prefix to infix expressions."
  (if (atom exp) exp
      (mapcar #'prefix->infix
              (if (binary-exp-p exp)
                  (list (exp-lhs exp) (exp-op exp) (exp-rhs exp))
                  exp))))

```

Here's an example of `solve-equations` in action, with a system of two equations. The reader should go through the trace, discovering which case was used at each call to `isolate`, and verifying that each step is accurate.

```

> (trace isolate solve)
(isolate solve)

> (solve-equations '((= (+ 3 4) (* (- 5 (+ 2 x)) 7))
                      (= (+ (* 3 x) y) 12)))
The equations to be solved are:
(3 + 4) = ((5 - (2 + X)) * 7)
((3 * X) + Y) = 12
(1 ENTER SOLVE: ((= (+ 3 4) (* (- 5 (+ 2 X)) 7))
                   (= (+ (* 3 X) Y) 12)) NIL)
(1 ENTER ISOLATE: (= (+ 3 4) (* (- 5 (+ 2 X)) 7)) X)
(2 ENTER ISOLATE: (= (* (- 5 (+ 2 X)) 7) (+ 3 4)) X)
(3 ENTER ISOLATE: (= (- 5 (+ 2 X)) (/ (+ 3 4) 7)) X)
(4 ENTER ISOLATE: (= (+ 2 X) (- 5 (/ (+ 3 4) 7))) X)
(5 ENTER ISOLATE: (= X (- (- 5 (/ (+ 3 4) 7)) 2)) X)
(5 EXIT ISOLATE: (= X (- (- 5 (/ (+ 3 4) 7)) 2)))
(4 EXIT ISOLATE: (= X (- (- 5 (/ (+ 3 4) 7)) 2)))

```

```

(3 EXIT ISOLATE: (= X (- (- 5 (/ (+ 3 4) 7)) 2)))
(2 EXIT ISOLATE: (= X (- (- 5 (/ (+ 3 4) 7)) 2)))
(1 EXIT ISOLATE: (= X (- (- 5 (/ (+ 3 4) 7)) 2)))
(2 ENTER SOLVE: ((= (+ (* 3 2) Y) 12)) ((= X 2)))
(1 ENTER ISOLATE: (= (+ (* 3 2) Y) 12) Y)
(2 ENTER ISOLATE: (= Y (- 12 (* 3 2))) Y)
(2 EXIT ISOLATE: (= Y (- 12 (* 3 2))))
(1 EXIT ISOLATE: (= Y (- 12 (* 3 2))))
(3 ENTER SOLVE: NIL ((= Y 6) (= X 2)))
(3 EXIT SOLVE: ((= Y 6) (= X 2)))
(2 EXIT SOLVE: ((= Y 6) (= X 2)))
(1 EXIT SOLVE: ((= Y 6) (= X 2)))
The solution is:
Y = 6
X = 2
NIL

```

Now let's tackle the format string "`~%~a~{~% ~{ ~a~}~}~%`" in print-equations. This may look like random gibberish, but there is actually sense behind it. format processes the string by printing each character, except that "`~`" indicates some special formatting action, depending on the following character. The combination "`~%`" prints a newline, and "`~a`" prints the next argument to format that has not been used yet. Thus the first four characters of the format string, "`~%~a`", print a newline followed by the argument header. The combination "`~{`" treats the corresponding argument as a list, and processes each element according to the specification between the "`~{`" and the next "`~}`". In this case, equations is a list of equations, so each one gets printed with a newline ("`~%`") followed by two spaces, followed by the processing of the equation itself as a list, where each element is printed in the "`~a`" format and preceded by a blank. The t given as the first argument to format means to print to the standard output; another output stream may be specified there.

One of the annoying minor holes in Lisp is that there is no standard convention on where to print newlines! In C, for example, the very first line of code in the reference manual is

```
printf("hello, world\n");
```

This makes it clear that newlines are printed *after* each line. This convention is so ingrained in the UNIX world that some UNIX programs will go into an infinite loop if the last line in a file is not terminated by a newline. In Lisp, however, the function print puts in a newline *before* the object to be printed, and a space after. Some Lisp programs carry the newline-before policy over to format, and others use the newline-after policy. This only becomes a problem when you want to combine two programs written under different policies. How did the two competing policies arise? In UNIX there was only one reasonable policy, because all input to the UNIX interpreter (the

shell) is terminated by newlines, so there is no need for a newline-before. In some Lisp interpreters, however, input can be terminated by a matching right parenthesis. In that case, a newline-before is needed, lest the output appear on the same line as the input.



**Exercise 7.1 [m]** Implement print-equations using only primitive printing functions such as terpri and princ, along with explicit loops.

## 7.3 Examples

Now we move on to examples, taken from Bobrow's thesis. In the first example, it is necessary to insert a "then" before the word "what" to get the right answer:

```
> (student '(If the number of customers Tom gets is twice the square of  
           20 % of the number of advertisements he runs ,,  
           and the number of advertisements is 45 ,,  
           then what is the number of customers Tom gets ?))
```

The equations to be solved are:

```
CUSTOMERS = (2 * (((20 / 100) * ADVERTISEMENTS) *  
                  ((20 / 100) * ADVERTISEMENTS)))  
ADVERTISEMENTS = 45  
WHAT = CUSTOMERS
```

The solution is:

```
WHAT = 162  
CUSTOMERS = 162  
ADVERTISEMENTS = 45  
NIL
```

Notice that our program prints the values for all variables it can solve for, while Bobrow's program only printed the values that were explicitly asked for in the text. This is an example of "more is less"—it may look impressive to print all the answers, but it is actually easier to do so than to decide just what answers should be printed. The following example is not solved correctly:

```
> (student '(The daily cost of living for a group is the overhead cost plus
   the running cost for each person times the number of people in
   the group .|. This cost for one group equals $ 100 .|,
   and the number of people in the group is 40 .|.
   If the overhead cost is 10 times the running cost .|,
   find the overhead and running cost for each person .|))
```

The equations to be solved are:

```
DAILY = (OVERHEAD + (RUNNING * PEOPLE))
COST = 100
PEOPLE = 40
OVERHEAD = (10 * RUNNING)
TO-FIND-1 = OVERHEAD
TO-FIND-2 = RUNNING
```

The solution is:

```
PEOPLE = 40
COST = 100
```

NIL

This example points out two important limitations of our version of student as compared to Bobrow's. The first problem is in naming of variables. The phrases "the daily cost of living for a group" and "this cost" are meant to refer to the same quantity, but our program gives them the names `daily` and `cost` respectively. Bobrow's program handled naming by first considering phrases to be the same only if they matched perfectly. If the resulting set of equations could not be solved, he would try again, this time considering phrases with words in common to be identical. (See the following exercises.)

The other problem is in our `solve` function. Assuming we got the variables equated properly, `solve` would be able to boil the set of equations down to two:

```
100 = (OVERHEAD + (RUNNING * 40))
OVERHEAD = (10 * RUNNING)
```

This is a set of two linear equations in two unknowns and has a unique solution at `RUNNING = 2`, `OVERHEAD = 20`. But our version of `solve` couldn't find this solution, since it looks for equations with one unknown. Here is another example that `student` handles well:

```
> (student '(Fran's age divided by Robin's height is one half Kelly's IQ .|
   Kelly's IQ minus 80 is Robin's height .|
   If Robin is 4 feet tall .| how old is Fran ?))
```

The equations to be solved are:

```
(FRAN / ROBIN) = (KELLY / 2)
(KELLY - 80) = ROBIN
ROBIN = 4
```

HOW = FRAN

The solution is:

HOW = 168

FRAN = 168

KELLY = 84

ROBIN = 4

NIL

But a slight variation leads to a problem:

```
> (student '(Fran's age divided by Robin's height is one half Kelly's IQ .|  
          Kelly's IQ minus 80 is Robin's height .|  
          If Robin is 0 feet tall .| how old is Fran ?))
```

The equations to be solved are:

(FRAN / ROBIN) = (KELLY / 2)

(KELLY - 80) = ROBIN

ROBIN = 0

HOW = FRAN

The solution is:

HOW = 0

FRAN = 0

KELLY = 80

ROBIN = 0

NIL

There is no valid solution to this problem, because it involves dividing by zero (Robin's height). But student is willing to transform the first equation into:

FRAN = ROBIN \* (KELLY / 2)

and then substitutes to get 0 for FRAN. Worse, dividing by zero could also come up inside eval:

```
> (student '(Fran's age times Robin's height is one half Kelly's IQ .|  
          Kelly's IQ minus 80 is Robin's height .|  
          If Robin is 0 feet tall .| how old is Fran ?))
```

The equations to be solved are:

(FRAN \* ROBIN) = (KELLY / 2)

(KELLY - 80) = ROBIN

ROBIN = 0

HOW = FRAN

>>Error: There was an attempt to divide a number by zero

However, one could claim that nasty examples with division by zero don't show up in algebra texts.

In summary, STUDENT behaves reasonably well, doing far more than the toy program ELIZA. STUDENT is also quite efficient; on my machine it takes less than one second for each of the prior examples. However, it could still be extended to have more powerful equation-solving capabilities. Its linguistic coverage is another matter. While one could add new patterns, such patterns are really just tricks, and don't capture the underlying structure of English sentences. That is why the STUDENT approach was abandoned as a research topic.

## 7.4 History and References

Bobrow's Ph.D. thesis contains a complete description of STUDENT. It is reprinted in Minsky 1968. Since then, there have been several systems that address the same task, with increased sophistication in both their mathematical and linguistic ability. Wong (1981) describes a system that uses its understanding of the problem to get a better linguistic analysis. Sterling et al. (1982) present a much more powerful equation solver, but it does not accept natural language input. Certainly Bobrow's language analysis techniques were not very sophisticated by today's measures. But that was largely the point: if you know that the language is describing an algebraic problem of a certain type, then you don't need to know very much linguistics to get the right answer most of the time.

## 7.5 Exercises



**Exercise 7.2 [h]** We said earlier that our program was unable to solve pairs of linear equations, such as:

$$\begin{aligned}100 &= (\text{OVERHEAD} + (\text{RUNNING} * 40)) \\ \text{OVERHEAD} &= (10 * \text{RUNNING})\end{aligned}$$

The original STUDENT could solve these equations. Write a routine to do so. You may assume there will be only two equations in two unknowns if you wish, or if you are more ambitious, you could solve a system of  $n$  linear equations with  $n$  unknowns.



**Exercise 7.3 [h]** Implement a version of Bobrow's variable-naming algorithm. Instead of taking the first word of each equation, create a unique symbol, and associate

with it the entire list of words. In the first pass, each nonequal list of words will be considered a distinct variable. If no solution is reached, word lists that share words in common are considered to be the same variable, and the solution is attempted again. For example, an input that contains the phrases “the rectangle’s width” and “the width of the rectangle” might assign these two phrases the variables v1 and v2. If an attempt to solve the problem yields no solutions, the program should realize that v1 and v2 have the words “rectangle” and “width” in common, and add the equation ( $= v1 v2$ ) and try again. Since the variables are arbitrary symbols, the printing routine should probably print the phrases associated with each variable rather than the variable itself.

-  **Exercise 7.4 [h]** The original STUDENT also had a set of “common knowledge” equations that it could use when necessary. These were mostly facts about conversion factors, such as (1 inch = 2.54 cm). Also included were equations like (distance equals rate times time), which could be used to solve problems like “If the distance from Anabru to Champaign is 10 miles and the time it takes Sandy to travel this distance is 2 hours, what is Sandy’s rate of speed?” Make changes to incorporate this facility. It probably only helps in conjunction with a solution to the previous exercise.
-  **Exercise 7.5 [h]** Change student so that it prints values only for those variables that are being asked for in the problem. That is, given the problem “X is 3. Y is 4. How much is X + Y?” it should not print values for X and Y.
-  **Exercise 7.6 [m]** Try STUDENT on the following examples. Make sure you handle special characters properly:
- The price of a radio is 69.70 dollars. If this price is 15% less than the marked price, find the marked price.
  - The number of soldiers the Russians have is one half of the number of guns they have. The number of guns they have is 7000. What is the number of soldiers they have?
  - If the number of customers Tom gets is twice the square of 20% of the number of advertisements he runs, and the number of advertisements is 45, and the profit Tom receives is 10 times the number of customers he gets, then what is the profit?
  - The average score is 73. The maximum score is 97. What is the square of the difference between the average and the maximum?
  - Tom is twice Mary’s age, and Jane’s age is half the difference between Mary and Tom. If Mary is 18 years old, how old is Jane?
  - What is  $4 + 5 * 14 / 7$ ?
  - $x \times b = c + d$ .  $b \times c = x$ .  $x = b + b$ .  $b = 5$ .

- ❑ **Exercise 7.7 [h]** Student's infix-to-prefix rules account for the priority of operators properly, but they don't handle associativity in the standard fashion. For example,  $(12 - 6 - 3)$  translates to  $(- 12 (- 6 3))$  or 9, when the usual convention is to interpret this as  $(- (- 12 6) 3)$  or 3. Fix student to handle this convention.
- ❑ **Exercise 7.8 [d]** Find a mathematically oriented domain that is sufficiently limited so that STUDENT can solve problems in it. The chemistry of solutions (calculating pH concentrations) might be an example. Write the necessary \*student-rules\*, and test the resulting program.
- ❑ **Exercise 7.9 [m]** Analyze the complexity of one-unknown and implement a more efficient version.
- ❑ **Exercise 7.10 [h]** Bobrow's paper on STUDENT (1968) includes an appendix that abstractly characterizes all the problems that his system can solve. Generate a similar characterization for this version of the program.

## 7.6 Answers

### Answer 7.1

```
(defun print-equations (header equations)
  (terpri)
  (princ header)
  (dolist (equation equations)
    (terpri)
    (princ " ")
    (dolist (x (prefix->infix equation))
      (princ " ")
      (princ x))))
```

**Answer 7.9** `one-unknown` is very inefficient because it searches each subcomponent of an expression twice. For example, consider the equation:

```
(= (+ (+ x 2) (+ 3 4)) (+ (+ 5 6) (+ 7 8)))
```

To decide if this has one unknown, `one-unknown` will call `no-unknown` on the left-hand side, and since it fails, call it again on the right-hand side. Although there are only eight atoms to consider, it ends up calling `no-unknown` 17 times and `one-unknown` 4 times. In general, for a tree of depth  $n$ , approximately  $2^n$  calls to `no-unknown` are made. This is clearly wasteful; there should be no need to look at each component more than once.

The following version uses an auxiliary function, `find-one-unknown`, that has an accumulator parameter, `unknown`. This parameter can take on three possible values: `nil`, indicating that no unknown has been found; or the single unknown that has been found so far; or the number `2` indicating that two unknowns have been found and therefore the final result should be `nil`. The function `find-one-unknown` has four cases: (1) If we have already found two unknowns, then return `2` to indicate this. (2) If the input expression is a nonatomic expression, then first look at its left-hand side for unknowns, and pass the result found in that side as the accumulator to a search of the right-hand side. (3) If the expression is an unknown, and if it is the second one found, return `2`; otherwise return the unknown itself. (4) If the expression is an atom that is not an unknown, then just return the accumulated result.

```
(defun one-unknown (exp)
  "Returns the single unknown in exp, if there is exactly one."
  (let ((answer (find-one-unknown exp nil)))
    ;; If there were two unknowns, return nil;
    ;; otherwise return the unknown (if there was one)
    (if (eql answer 2)
        nil
        answer)))

(defun find-one-unknown (exp unknown)
  "Assuming UNKNOWN is the unknown(s) found so far, decide
  if there is exactly one unknown in the entire expression."
  (cond ((eql unknown 2) 2)
        ((exp-p exp)
         (find-one-unknown
          (exp-rhs exp)
          (find-one-unknown (exp-lhs exp) unknown)))
        ((unknown-p exp)
         (if unknown
             2
             exp))
        (t unknown)))
```

## CHAPTER 8

---

# Symbolic Mathematics: A Simplification Program

*Our life is frittered away by detail. . . .  
Simplify, simplify.*

—Henry David Thoreau, *Walden* (1854)

“Symbolic mathematics” is to numerical mathematics as algebra is to arithmetic: it deals with variables and expressions rather than just numbers. Computers were first developed primarily to solve arithmetic problems: to add up large columns of numbers, to multiply many-digit numbers, to solve systems of linear equations, and to calculate the trajectories of ballistics. Encouraged by success in these areas, people hoped that computers could also be used on more complex problems; to differentiate or integrate a mathematical expression and come up with another expression as the answer, rather than just a number. Several programs were developed along these lines in the 1960s and 1970s. They were used primarily by professional mathematicians and physicists with access to large mainframe computers. Recently, programs like MATHLAB, DERIVE, and MATHEMATICA have given these capabilities to the average personal computer user.

It is interesting to look at some of the history of symbolic algebra, beginning in 1963 with SAINT, James Slagle's program to do symbolic integration. Originally, SAINT was heralded as a triumph of AI. It used general problem-solving techniques, similar in kind to GPS, to search for solutions to difficult problems. The program worked its way through an integration problem by choosing among the techniques known to it and backing up when an approach failed to pan out. SAINT's behavior on such problems was originally similar to (and eventually much better than) the performance of undergraduate calculus students.

Over time, the AI component of symbolic integration began to disappear. Joel Moses implemented a successor to SAINT called SIN. It used many of the same techniques, but instead of relying on search to find the right combination of techniques, it had additional mathematical knowledge that led it to pick the right technique at each step, without any provision for backing up and trying an alternative. SIN solved more problems and was much faster than SAINT, although it was not perfect: it still occasionally made the wrong choice and failed to solve a problem it could have.

By 1970, the mathematician R. Risch and others developed algorithms for indefinite integration of any expression involving algebraic, logarithmic, or exponential extensions of rational functions. In other words, given a "normal" function, the Risch algorithm will return either the indefinite integral of the function or an indication that no closed-form integral is possible in terms of elementary functions. Such work effectively ended the era of considering integration as a problem in search.

SIN was further refined, merged with parts of the Risch algorithm, and put into the evolving MACSYMA<sup>1</sup> program. For the most part, refinement of MACSYMA consisted of the incorporation of new algorithms. Few heuristics of any sort survive. Today MACSYMA is no longer considered an AI program. It is used daily by scientists and mathematicians, while ELIZA and STUDENT are now but historical footnotes.

With ELIZA and STUDENT we were able to develop miniature programs that duplicated most of the features of the original. We won't even try to develop a program worthy of the name MACSYMA; instead we will settle for a modest program to do symbolic simplification, which we will call (simply) *simplifier*. Then, we will extend *simplifier* to do differentiation, and some integration problems. The idea is that given an expression like  $(2 - 1)x + 0$ , we want the program to compute the simplified form  $x$ .

According to the *Mathematics Dictionary* (James and James 1949), the word "simplified" is "probably the most indefinite term used seriously in mathematics." The problem is that "simplified" is relative to what you want to use the expression for next. Which is simpler,  $x^2 + 3x + 2$  or  $(x + 1)(x + 2)$ ? The first makes it easier to

---

<sup>1</sup>MACSYMA is the Project MAC SYMbolic MAthematics program. Project MAC is the MIT research organization that was the precursor of MIT's Laboratory for Computer Science. MAC stood either for Machine-Aided Cognition or Multiple-Access Computer, according to one of their annual reports. The cynical have claimed that MAC really stood for Man Against Computer.

integrate or differentiate, the second easier to find roots. We will be content to limit ourselves to “obvious” simplifications. For example,  $x$  is almost always preferable to  $1x + 0$ .

## 8.1 Converting Infix to Prefix Notation

We will represent simplifications as a list of rules, much like the rules for STUDENT and ELIZA. But since each simplification rule is an algebraic equation, we will store each one as an exp rather than as a rule. To make things more legible, we will write each expression in infix form, but store them in the prefix form expected by exp. This requires an infix->prefix function to convert infix expressions into prefix notation. We have a choice as to how general we want our infix notation to be. Consider:

```
((((a * (x ^ 2)) + (b * x)) + c)
(a * x ^ 2 + b * x + c)
(a x ^ 2 + b x + c)
a x^2 + b*x+c
```

The first is fully parenthesized infix, the second makes use of operator precedence (multiplication binds tighter than addition and is thus performed first), and the third makes use of implicit multiplication as well as operator precedence. The fourth requires a lexical analyzer to break Lisp symbols into pieces.

Suppose we only wanted to handle the fully parenthesized case. To write infix->prefix, one might first look at prefix->infix (on page 228) trying to adapt it to our new purposes. In doing so, the careful reader might discover a surprise: infix->prefix and prefix->infix are in fact the exact same function! Both leave atoms unchanged, and both transform three-element lists by swapping the exp-op and exp-lhs. Both apply themselves recursively to the (possibly rearranged) input list. Once we discover this fact, it would be tempting to avoid writing infix->prefix, and just call prefix->infix instead. Avoid this temptation at all costs. Instead, define infix->prefix as shown below. The intent of your code will be clearer:

```
(defun infix->prefix (infix-exp)
  "Convert fully parenthesized infix-exp to a prefix expression"
  ;;= Don't use this version for non-fully parenthesized exps!
  (prefix->infix infix-exp))
```

As we saw above, fully parenthesized infix can be quite ugly, with all those extra parentheses, so instead we will use operator precedence. There are a number of ways of doing this, but the easiest way for us to proceed is to use our previously defined tool rule-based-translator and its subtool, pat-match. Note that the third

clause of `infix->prefix`, the one that calls `rule-based-translator` is unusual in that it consists of a single expression. Most `cond`-clauses have two expressions: a test and a result, but ones like this mean, “Evaluate the test, and if it is non-nil, return it. Otherwise go on to the next clause.”

```
(defun infix->prefix (exp)
  "Translate an infix expression into prefix notation."
  ;;= Note we cannot do implicit multiplication in this system
  (cond ((atom exp) exp)
        ((= (length exp) 1) (infix->prefix (first exp)))
        ((rule-based-translator exp *infix->prefix-rules*
          :rule-if #'rule-pattern :rule-then #'rule-response
          :action
          #'(lambda (bindings response)
              (sublis (mapcar
                        #'(lambda (pair)
                            (cons (first pair)
                                  (infix->prefix (rest pair)))))
                        bindings)
              response)))
        ((symbolp (first exp))
         (list (first exp) (infix->prefix (rest exp))))
        (t (error "Illegal exp"))))
```

Because we are doing mathematics in this chapter, we adopt the mathematical convention of using certain one-letter variables, and redefine `variable-p` so that variables are only the symbols `m` through `z`.

```
(defun variable-p (exp)
  "Variables are the symbols M through Z."
  ;;= put x,y,z first to find them a little faster
  (member exp '(x y z m n o p q r s t u v w))

  (pat-match-abbrev 'x+ '(?+ x))
  (pat-match-abbrev 'y+ '(?+ y))

  (defun rule-pattern (rule) (first rule))
  (defun rule-response (rule) (second rule))
```

```
(defparameter *infix->prefix-rules*
  (mapcar #'expand-pat-match-abbrev
    '(((x+ = y+) (= x y))
      ((- x+) (- x))
      ((+ x+) (+ x))
      ((x+ + y+) (+ x y))
      ((x+ - y+) (- x y))
      ((x+ * y+) (* x y))
      ((x+ / y+) (/ x y))
      ((x+ ^ y+) (^ x y))))
  "A list of rules, ordered by precedence."))

```

## 8.2 Simplification Rules

Now we are ready to define the simplification rules. We use the definition of the data types rule and exp (page 221) and prefix->infix (page 228) from STUDENT. They are repeated here:

```
(defstruct (rule (:type list)) pattern response)
(defstruct (exp (:type list)
  (:constructor mkexp (lhs op rhs)))
  op lhs rhs)
(defun exp-p (x) (consp x))
(defun exp-args (x) (rest x))

(defun prefix->infix (exp)
  "Translate prefix to infix expressions."
  (if (atom exp) exp
    (mapcar #'prefix->infix
      (if (binary-exp-p exp)
        (list (exp-lhs exp) (exp-op exp) (exp-rhs exp))
        exp)))))

(defun binary-exp-p (x)
  (and (exp-p x) (= (length (exp-args x)) 2)))
```

We also use rule-based-translator (page 188) once again, this time on a list of simplification rules. A reasonable list of simplification rules is shown below. This list covers the four arithmetic operators, addition, subtraction, multiplication, and division, as well as exponentiation (raising to a power), denoted by the symbol “ $\wedge$ ”.

Again, it is important to note that the rules are ordered, and that later rules will be applied only when earlier rules do not match. So, for example,  $0 / 0$  simplifies to

undefined, and not to 1 or 0, because the rule for  $0 / 0$  comes before the other rules. See exercise 8.8 for a more complete treatment of this.

```
(defparameter *simplification-rules* (mapcar #'infix->prefix '(  
  (x + 0 = x)  
  (0 + x = x)  
  (x + x = 2 * x)  
  (x - 0 = x)  
  (0 - x = - x)  
  (x - x = 0)  
  (- - x = x)  
  (x * 1 = x)  
  (1 * x = x)  
  (x * 0 = 0)  
  (0 * x = 0)  
  (x * x = x ^ 2)  
  (x / 0 = undefined)  
  (0 / x = 0)  
  (x / 1 = x)  
  (x / x = 1)  
  (0 ^ 0 = undefined)  
  (x ^ 0 = 1)  
  (0 ^ x = 0)  
  (1 ^ x = 1)  
  (x ^ 1 = x)  
  (x ^ -1 = 1 / x)  
  (x * (y / x) = y)  
  ((y / x) * x = y)  
  ((y * x) / x = y)  
  ((x * y) / x = y)  
  (x + - x = 0)  
  ((- x) + x = 0)  
  (x + y - x = y)  
 )))  
  
(defun ^ (x y) "Exponentiation" (expt x y))
```

We are now ready to go ahead and write the simplifier. The main function, `simplifier`, will repeatedly print a prompt, read an input, and print it in simplified form. Input and output is in infix and the computation is in prefix, so we need to convert accordingly; the function `simp` does this, and the function `simplify` takes care of a single prefix expression. It is summarized in figure 8.1.

	<b>Top-Level Functions</b>
simplifier	A read-simplify-print loop.
simp	Simplify an infix expression.
simplify	Simplify a prefix expression.
	<b>Special Variables</b>
*infix->prefix-rules*	Rules to translate from infix to prefix.
*simplification-rules*	Rules to simplify an expression.
	<b>Data Types</b>
exp	A prefix expression.
	<b>Auxiliary Functions</b>
simplify-exp	Simplify a non-atomic prefix expression.
infix->prefix	Convert infix to prefix notation.
variable-p	The symbols m through z are variables.
^	An alias for expt, exponentiation.
evaluable	Decide if an expression can be numerically evaluated.
simp-rule	Transform a rule into proper format.
length=1	Is the argument a list of length 1?
	<b>Previous Functions</b>
pat-match	Match pattern against an input. (p. 180)
rule-based-translator	Apply a set of rules. (p. 189)
pat-match-abbrev	Define an abbreviation for use in pat-match.

Figure 8.1: Glossary for the Simplifier

Here is the program:

```
(defun simplifier ()
  "Read a mathematical expression, simplify it, and print the result."
  (loop
    (print 'simplifier)
    (print (simp (read)))))

(defun simp (inf) (prefix->infix (simplify (infix->prefix inf)))))

(defun simplify (exp)
  "Simplify an expression by first simplifying its components."
  (if (atom exp) exp
      (simplify-exp (mapcar #'simplify exp)))))

(defun simplify-exp (exp)
  "Simplify using a rule, or by doing arithmetic."
  (cond ((rule-based-translator exp *simplification-rules*
        :rule-if #'exp-lhs :rule-then #'exp-rhs
        :action #'(lambda (bindings response)
                    (simplify (sublis bindings response)))))
        ((evaluable exp) (eval exp))
        (t exp)))
```

---

```
(defun evaluable (exp)
  "Is this an arithmetic expression that can be evaluated?"
  (and (every #'numberp (exp-args exp))
       (or (member (exp-op exp) '(+ - * /))
           (and (eq (exp-op exp) '^)
                (integerp (second (exp-args exp)))))))
```

The function `simplify` assures that any compound expression will be simplified by first simplifying the arguments and then calling `simplify-exp`. This latter function searches through the simplification rules, much like `use-eliza-rules` and `translate-to-expression`. When it finds a match, `simplify-exp` substitutes in the proper variable values and calls `simplify` on the result. `simplify-exp` also has the ability to call `eval` to simplify an arithmetic expression to a number. As in STUDENT, it is for the sake of this `eval` that we require expressions to be represented as lists in prefix notation. Numeric evaluation is done *after* checking the rules so that the rules can intercept expressions like  $(/ 1 0)$  and simplify them to `undefined`. If we did the numeric evaluation first, these expressions would yield an error when passed to `eval`. Because Common Lisp supports arbitrary precision rational numbers (fractions), we are guaranteed there will be no round-off error, unless the input explicitly includes inexact (floating-point) numbers. Notice that we allow computations involving the four arithmetic operators, but exponentiation is only allowed if the exponent is an integer. That is because expressions like  $(^ 4 1/2)$  are not guaranteed to return 2 (the exact square root of 4); the answer might be 2.0 (an inexact number). Another problem is that  $-2$  is also a square root of 4, and in some contexts it is the correct one to use.

The following trace shows some examples of the simplifier in action. First we show that it can be used as a calculator; then we show more advanced problems.

```
> (simplifier)
SIMPLIFIER> (2 + 2)
4
SIMPLIFIER> (5 * 20 + 30 + 7)
137
SIMPLIFIER> (5 * x - (4 + 1) * x)
0
SIMPLIFIER> (y / z * (5 * x - (4 + 1) * x))
0
SIMPLIFIER> ((4 - 3) * x + (y / y - 1) * z)
X
SIMPLIFIER> (1 * f(x) + 0)
(F X)
SIMPLIFIER> (3 * 2 * X)
(3 * (2 * X))
SIMPLIFIER> [Abort]
>
```

Here we have terminated the loop by hitting the abort key on the terminal. (The details of this mechanism varies from one implementation of Common Lisp to another.) The simplifier seems to work fairly well, although it errs on the last example:  $(3 * (2 * X))$  should simplify to  $(6 * X)$ . In the next section, we will correct that problem.

### 8.3 Associativity and Commutativity

We could easily add a rule to rewrite  $(3 * (2 * X))$  as  $((3 * 2) * X)$  and hence  $(6 * X)$ . The problem is that this rule would also rewrite  $(X * (2 * 3))$  as  $((X * 2) * 3)$ , unless we had a way to limit the rule to apply only when it would group numbers together. Fortunately, pat-match does provide just this capability, with the `?is` pattern. We could write this rule:

$$((?is\ n\ numberp)\ * ((?is\ m\ numberp)\ * x)) = ((n\ * m)\ * x))$$

This transforms  $(3 * (2 * x))$  into  $((3 * 2) * x)$ , and hence into  $(6 * x)$ . Unfortunately, the problem is not as simple as that. We also want to simplify  $((2 * x) * (y * 3))$  to  $(6 * (x * y))$ . We can do a better job of gathering numbers together by adopting three conventions. First, make numbers first in products: change  $x * 3$  to  $3 * x$ . Second, combine numbers in an outer expression with a number in an inner expression: change  $3 * (5 * x)$  to  $(3 * 5) * x$ . Third, move numbers out of inner expressions whenever possible: change  $(3 * x) * y$  to  $3 * (x * y)$ . We adopt similar conventions for addition, except that we prefer numbers last there:  $x + 1$  instead of  $1 + x$ .

```
(setf *simplification-rules*
      (append *simplification-rules* (mapcar #'simp-rule
        '((s * n = n * s)
          (n * (m * x) = (n * m) * x)
          (x * (n * y) = n * (x * y))
          ((n * x) * y = n * (x * y))
          (n + s = s + n)
          ((x + m) + n = x + n + m)
          (x + (y + n) = (x + y) + n)
          ((x + n) + y = (x + y) + n)))))
```

With the new rules in place, we are ready to try again. For some problems we get just the right answers:

```
> (simplifier)
SIMPLIFIER> (3 * 2 * x)
(6 * x)
SIMPLIFIER> (2 * x * x * 3)
(6 * (x ^ 2))
SIMPLIFIER> (2 * x * 3 * y * 4 * z * 5 * 6)
(720 * (x * (y * z)))
SIMPLIFIER> (3 + x + 4 + x)
((2 * x) + 7)
SIMPLIFIER> (2 * x * 3 * x * 4 * (1 / x) * 5 * 6)
(720 * x)
```

Unfortunately, there are other problems that aren't simplified properly:

```
SIMPLIFIER> (3 + x + 4 - x)
((x + (4 - x)) + 3)
SIMPLIFIER> (x + y + y + x)
(x + (y + (y + x)))
SIMPLIFIER> (3 * x + 4 * x)
((3 * x) + (4 * x))
```

We will return to these problems in section 8.5.

-  **Exercise 8.1** Verify that the set of rules just prior does indeed implement the desired conventions, and that the conventions have the proper effect, and always terminate. As an example of a potential problem, what would happen if we used the rule ( $x * n = n * x$ ) instead of the rule ( $s * n = n * s$ )?

## 8.4 Logs, Trig, and Differentiation

In the previous section, we restricted ourselves to the simple arithmetic functions, so as not to intimidate those who are a little leery of complex mathematics. In this section, we add a little to the mathematical complexity, without having to alter the program itself one bit. Thus, the mathematically shy can safely skip to the next section without feeling they are missing any of the fun.

We start off by representing some elementary properties of the logarithmic and trigonometric functions. The new rules are similar to the “zero and one” rules we needed for the arithmetic operators, except here the constants  $e$  and  $\pi$  ( $e = 2.71828\dots$  and  $\pi = 3.14159\dots$ ) are important in addition to 0 and 1. We also throw in some rules relating logs and exponents, and for sums and differences of logs. The rules assume that complex numbers are not allowed. If they were,  $\log e^x$  (and even  $x^y$ ) would have multiple values, and it would be wrong to arbitrarily choose one of these values.

```
(setf *simplification-rules*
  (append *simplification-rules* (mapcar #'simp-rule '(
    (log 1          = 0)
    (log 0          = undefined)
    (log e          = 1)
    (sin 0          = 0)
    (sin pi         = 0)
    (cos 0          = 1)
    (cos pi         = -1)
    (sin(pi / 2)   = 1)
    (cos(pi / 2)   = 0)
    (log (e ^ x)   = x)
    (e ^ (log x)   = x)
    ((x ^ y) * (x ^ z) = x ^ (y + z))
    ((x ^ y) / (x ^ z) = x ^ (y - z))
    (log x + log y = log(x * y))
    (log x - log y = log(x / y))
    ((sin x) ^ 2 + (cos x) ^ 2 = 1)
  ))))
```

Now we would like to go a step further and extend the system to handle differentiation. This is a favorite problem, and one which has historical significance: in the summer of 1958 John McCarthy decided to investigate differentiation as an interesting symbolic computation problem, which was difficult to express in the primitive programming languages of the day. This investigation led him to see the importance of functional arguments and recursive functions in the field of symbolic computation. For example, McCarthy invented what we now call `mapcar` to express the idea that the derivative of a sum is the sum of the derivative function applied to each argument. Further work led McCarthy to the publication in October 1958 of MIT

AI Lab Memo No. 1: "An Algebraic Language for the Manipulation of Symbolic Expressions," which defined the precursor of Lisp.

In McCarthy's work and in many subsequent texts you can see symbolic differentiation programs with a simplification routine tacked on the end to make the output more readable. Here, we take the opposite approach: the simplification routine is central, and differentiation is handled as just another operator, with its own set of simplification rules. We will require a new infix-to-prefix translation rule. While we're at it, we'll add a rule for indefinite integration as well, although we won't write simplification rules for integration yet. Here are the new notations:

math	infix	prefix
$dy/dx$	$d y / d x$	$(d y x)$
$\int y dx$	$\text{Int } y \text{ } d x$	$(\text{int } y x)$

And here are the necessary infix-to-prefix rules:

```
(defparameter *infix->prefix-rules*
  (mapcar #'expand-pat-match-abbrev
    '((((x+ = y+) (= x y))
       ((- x+) (- x))
       ((+ x+) (+ x))
       ((x+ + y+) (+ x y))
       ((x+ - y+) (- x y))
       ((d y+ / d x) (d y x)) ;*** New rule
       ((Int y+ d x) (int y x)) ;*** New rule
       ((x+ * y+) (* x y))
       ((x+ / y+) (/ x y))
       ((x+ ^ y+) (^ x y)))))
```

Since the new rule for differentiation occurs before the rule for division, there won't be any confusion with a differential being interpreted as a quotient. On the other hand, there is a potential problem with integrals that contain  $d$  as a variable. The user can always avoid the problem by using  $(d)$  instead of  $d$  inside an integral.

Now we augment the simplification rules, by copying a differentiation table out of a reference book:

```
(setf *simplification-rules*
  (append *simplification-rules* (mapcar #'simp-rule '(
    (d x / d x      = 1)
    (d (u + v) / d x = (d u / d x) + (d v / d x))
    (d (u - v) / d x = (d u / d x) - (d v / d x))
    (d (- u) / d x = - (d u / d x))
    (d (u * v) / d x = u * (d v / d x) + v * (d u / d x))
    (d (u / v) / d x = (v * (d u / d x) - u * (d v / d x))
                           / v ^ 2)
```

```
(d (u ^ n) / d x = n * u ^ (n - 1) * (d u / d x))
(d (u ^ v) / d x = v * u ^ (v - 1) * (d u / d x)
+ u ^ v * (log u) * (d v / d x))
(d (log u) / d x = (d u / d x) / u)
(d (sin u) / d x = (cos u) * (d u / d x))
(d (cos u) / d x = - (sin u) * (d u / d x))
(d (e ^ u) / d x = (e ^ u) * (d u / d x))
(d u / d x      = 0))))))
```

We have added a default rule,  $(d u / d x = 0)$ ; this should only apply when the expression  $u$  is free of the variable  $x$  (that is, when  $u$  is not a function of  $x$ ). We could use `?if` to check this, but instead we rely on the fact that differentiation is closed over the list of operators described here—as long as we don’t introduce any new operators, the answer will always be correct. Note that there are two rules for exponentiation, one for the case when the exponent is a number, and one when it is not. This was not strictly necessary, as the second rule covers both cases, but that was the way the rules were written in the table of differentials I consulted, so I left both rules in.

```
SIMPLIFIER> (d (x + x) / d x)
2
SIMPLIFIER> (d (a * x ^ 2 + b * x + c) / d x)
((2 * (A * X)) + B)
SIMPLIFIER> (d ((a * x ^ 2 + b * x + c) / x) / d x)
(((A * (X ^ 2)) + ((B * X) + C)) - (X * ((2 * (A * X)) + B)))
/ (X ^ 2)
SIMPLIFIER> (log ((d (x + x) / d x) / 2))
0
SIMPLIFIER> (log(x + x) - log x)
(LOG 2)
SIMPLIFIER> (x ^ cos p1)
(1 / X)
SIMPLIFIER> (d (3 * x + (cos x) / x) / d x)
(((COS X) - (X * (- (SIN X)))) / (X ^ 2)) + 3
SIMPLIFIER> (d ((cos x) / x) / d x)
(((COS X) - (X * (- (SIN X)))) / (X ^ 2))
SIMPLIFIER> (d (3 * x ^ 2 + 2 * x + 1) / d x)
((6 * X) + 2)
SIMPLIFIER> (sin(x + x) ^ 2 + cos(d x ^ 2 / d x) ^ 2)
1
SIMPLIFIER> (sin(x + x) * sin(d x ^ 2 / d x) +
cos(2 * x) * cos(x * d 2 * y / d y))
1
```

The program handles differentiation problems well and is seemingly clever in its use of the identity  $\sin^2 x + \cos^2 x = 1$ .

## 8.5 Limits of Rule-Based Approaches

In this section we return to some examples that pose problems for the simplifier. Here is a simple one:

```
SIMPLIFIER> (x + y + y + x) ⇒ (X + (Y + (Y + X)))
```

We would prefer  $2 * (x + y)$ . The problem is that, although we went to great trouble to group numbers together, there was no effort to group non-numbers. We could write rules of the form:

$$\begin{aligned} (y + (y + x)) &= (2 * y) + x \\ (y + (x + y)) &= (2 * y) + x \end{aligned}$$

These would work for the example at hand, but they would not work for  $(x + y + z + y + x)$ . For that we would need more rules:

$$\begin{aligned} (y + (z + (y + x))) &= (2 * y) + x + z \\ (y + (z + (x + y))) &= (2 * y) + x + z \\ (y + ((y + x) + z)) &= (2 * y) + x + z \\ (y + ((x + y) + z)) &= (2 * y) + x + z \end{aligned}$$

To handle all the cases, we would need an infinite number of rules. The pattern-matching language is not powerful enough to express this succinctly. It might help if nested sums (and products) were unnested; that is, if we allowed  $+$  to take an arbitrary number of arguments instead of just one. Once the arguments are grouped together, we could sort them, so that, say, all the  $y$ s appear before  $z$  and after  $x$ . Then like terms could be grouped together. We have to be careful, though. Consider these examples:

```
SIMPLIFIER> (3 * x + 4 * x)
((3 * X) + (4 * X))
SIMPLIFIER> (3 * x + y + x + 4 * x)
((3 * X) + (Y + (X + (4 * X))))
```

We would want  $(3 * x)$  to sort to the same place as  $x$  and  $(4 * x)$  so that they could all be combined to  $(8 * x)$ . In chapter 15, we develop a new version of the program that handles this problem.

## 8.6 Integration

So far, the algebraic manipulations have been straightforward. There is a direct algorithm for computing the derivative of every expression. When we consider integrals, or antiderivatives,<sup>2</sup> the picture is much more complicated. As you may recall from freshman calculus, there is a fine art to computing integrals. In this section, we try to see how far we can get by encoding just a few of the many tricks available to the calculus student.

The first step is to recognize that entries in the simplification table will not be enough. Instead, we will need an algorithm to evaluate or “simplify” integrals. We will add a new case to `simplify-exp` to check each operator to see if it has a simplification function associated with it. These simplification functions will be associated with operators through the functions `set-simp-fn` and `simp-fn`. If an operator does have a simplification function, then that function will be called instead of consulting the simplification rules. The simplification function can elect not to handle the expression after all by returning `nil`, in which case we continue with the other simplification methods.

```
(defun simp-fn (op) (get op 'simp-fn))
(defun set-simp-fn (op fn) (setf (get op 'simp-fn) fn))

(defun simplify-exp (exp)
  "Simplify using a rule, or by doing arithmetic,
  or by using the simp function supplied for this operator."
  (cond ((simplify-by-fn exp) ;***
         ((rule-based-translator exp *simplification-rules*
           :rule-if #'exp-lhs :rule-then #'exp-rhs
           :action #'(lambda (bindings response)
                       (simplify (sublis bindings response)))))

         ((evaluable exp) (eval exp))
         (t exp)))
        ((defun simplify-by-fn (exp)
          "If there is a simplification fn for this exp,
          and if applying it gives a non-null result,
          then simplify the result and return that."
          (let* ((fn (simp-fn (exp-op exp))))
            (result (if fn (funcall fn exp)))))
         (if (null result)
             nil
             (simplify result))))
```

Freshman calculus classes teach a variety of integration techniques. Fortunately, one technique—the derivative-divides technique—can be adopted to solve most of the

---

<sup>2</sup>The term antiderivative is more correct, because of branch point problems.

problems that come up at the freshman calculus level, perhaps 90% of the problems given on tests. The basic rule is:

$$\int f(x) dx = \int f(u) \frac{du}{dx} dx.$$

As an example, consider  $\int x \sin(x^2) dx$ . Using the substitution  $u = x^2$ , we can differentiate to get  $du/dx = 2x$ . Then by applying the basic rule, we get:

$$\int x \sin(x^2) dx = \frac{1}{2} \int \sin(u) \frac{du}{dx} dx = \frac{1}{2} \int \sin(u) du.$$

Assume we have a table of integrals that includes the rule  $\int \sin(x) dx = -\cos(x)$ . Then we can get the final answer:

$$-\frac{1}{2} \cos(x^2).$$

Abstracting from this example, the general algorithm for integrating an expression  $y$  with respect to  $x$  is:

1. Pick a factor of  $y$ , calling it  $f(u)$ .
2. Compute the derivative  $du/dx$ .
3. Divide  $y$  by  $f(u) \times du/dx$ , calling the quotient  $k$ .
4. If  $k$  is a constant (with respect to  $x$ ), then the result is  $k \int f(u) du$ .

This algorithm is nondeterministic, as there may be many factors of  $y$ . In our example,  $f(u) = \sin(x^2)$ ,  $u = x^2$ , and  $du/dx = 2x$ . So  $k = \frac{1}{2}$ , and the answer is  $-\frac{1}{2} \cos(x^2)$ .

The first step in implementing this technique is to make sure that division is done correctly. We need to be able to pick out the factors of  $y$ , divide expressions, and then determine if a quotient is free of  $x$ . The function `factorize` does this. It keeps a list of factors and a running product of constant factors, and augments them with each call to the local function `fac`.

```
(defun factorize (exp)
  "Return a list of the factors of exp^n,
  where each factor is of the form (^ y n)."
  (let ((factors nil)
        (constant 1))
    (labels
      ((fac (x n)
            (cond
              ((numberp x)
               (setf constant (* constant (expt x n))))
              ((starts-with x '**)
               (fac (exp-lhs x) n)
               (fac (exp-rhs x) n))
              ((starts-with x '/)
               (fac (exp-lhs x) n)
               (fac (exp-rhs x) (- n)))
              ((and (starts-with x '-')
                     (length=1 (exp-args x)))
               (setf constant (- constant))
               (fac (exp-lhs x) n))
              ((and (starts-with x '^)
                     (numberp (exp-rhs x)))
               (fac (exp-lhs x) (* n (exp-rhs x)))))
              (t (let ((factor (find x factors :key #'exp-lhs
                                         :test #'equal)))
                  (if factor
                      (incf (exp-rhs factor) n)
                      (push '(^ ,x ,n) factors)))))))
        ;;= Body of factorize:
        (fac exp 1)
        (case constant
          (0 '(^)(^ 0 1))
          (1 factors)
          (t '(^)(^ ,constant 1) .,factors))))))

```

`factorize` maps from an expression to a list of factors, but we also need `unfactorize` to turn a list back into an expression:

```
(defun unfactorize (factors)
  "Convert a list of factors back into prefix form."
  (cond ((null factors) 1)
        ((length=1 factors) (first factors))
        (t '(* ,(first factors) ,(unfactorize (rest factors))))))
```

The derivative-divides method requires a way of dividing two expressions. We do this by factoring each expression and then dividing by cancelling factors. There may be cases where, for example, two factors in the numerator could be multiplied together

to cancel a factor in the denominator, but this possibility is not considered. It turns out that most problems from freshman calculus do not require such sophistication.

```
(defun divide-factors (numer denom)
  "Divide a list of factors by another, producing a third."
  (let ((result (mapcar #'copy-list numer)))
    (dolist (d denom)
      (let ((factor (find (exp-lhs d) result :key #'exp-lhs
                           :test #'equal)))
        (if factor
            (decf (exp-rhs factor) (exp-rhs d))
            (push '(^ ,(exp-lhs d) ,(- (exp-rhs d)) result))))))
    (delete 0 result :key #'exp-rhs)))
```

Finally, the predicate `free-of` returns true if an expression does not have any occurrences of a particular variable in it.

```
(defun free-of (exp var)
  "True if expression has no occurrence of var."
  (not (find-anywhere var exp)))

(defun find-anywhere (item tree)
  "Does item occur anywhere in tree? If so, return it."
  (cond ((eql item tree) tree)
        ((atom tree) nil)
        ((find-anywhere item (first tree)))
        ((find-anywhere item (rest tree)))))
```

In `factorize` we made use of the auxiliary function `length=1`. The function call `(length=1 x)` is faster than `(= (length x) 1)` because the latter has to compute the length of the whole list, while the former merely has to see if the list has a `rest` element or not.

```
(defun length=1 (x)
  "Is X a list of length 1?"
  (and (consp x) (null (rest x))))
```

Given these preliminaries, the function `integrate` is fairly easy. We start with some simple cases for integrating sums and constant expressions. Then, we factor the expression and split the list of factors into two: a list of constant factors, and a list of factors containing  $x$ . (This is done with `partition-if`, a combination of `remove-if` and `remove-if-not`.) Finally, we call `deriv-divides`, giving it a chance with each of the factors. If none of them work, we return an expression indicating that the integral is unknown.

```

(defun integrate (exp x)
  ;; First try some trivial cases
  (cond
    ((free-of exp x) '(* ,exp x))           ; Int c dx = c*x
    ((starts-with exp '+)                   ; Int f + g =
     '(+ ,(integrate (exp-lhs exp) x)      ;   Int f + Int g
          ,(integrate (exp-rhs exp) x)))
    ((starts-with exp '-)
     (ecase (length (exp-args exp))
       (1 (integrate (exp-lhs exp) x))      ; Int - f = - Int f
       (2 '(- ,(integrate (exp-lhs exp) x) ; Int f - g =
              ,(integrate (exp-rhs exp) x)))))) ; Int f - Int g
    ;; Now move the constant factors to the left of the integral
    ((multiple-value-bind (const-factors x-factors)
        (partition-if #'(lambda (factor) (free-of factor x))
                      (factorize exp))
        (simplify
         '(* ,(unfactorize const-factors)
              ;; And try to integrate:
              ,(cond ((null x-factors) x)
                     ((some #'(lambda (factor)
                               (deriv-divides factor x-factors x))
                            x-factors))
                     ;; <other methods here>
                     (t '(int? ,(unfactorize x-factors) ,x))))))))
      (defun partition-if (pred list)
        "Return 2 values: elements of list that satisfy pred,
        and elements that don't."
        (let ((yes-list nil)
              (no-list nil))
          (dolist (item list)
            (if (funcall pred item)
                (push item yes-list)
                (push item no-list)))
          (values (nreverse yes-list) (nreverse no-list))))
```

Note that the place in `integrate` where other techniques could be added is marked. We will only implement the derivative-divides method. It turns out that the function is a little more complicated than the simple four-step algorithm outlined before:

```
(defun deriv-divides (factor factors x)
  (assert (starts-with factor '^))
  (let* ((u (exp-lhs factor)) ; factor = u^n
         (n (exp-rhs factor))
         (k (divide-factors
              factors (factorize '(* ,factor ,(deriv u x))))))
    (cond ((free-of k x)
           (;; Int k*u^n*du/dx dx = k*Int u^n du
           ;; = k*u^(n+1)/(n+1) for n /= -1
           ;; = k*log(u) for n = -1
           (if (= n -1)
               '(* ,(unfactorize k) (log ,u))
               '/ (* ,(unfactorize k) (^ ,u ,(+ n 1)))
                   ,(+ n 1)))
           ((and (= n 1) (in-integral-table? u))
            (;; Int y'*f(y) dx = Int f(y) dy
            (let ((k2 (divide-factors
                      factors
                      (factorize '(* ,u ,(deriv (exp-lhs u) x))))))
              (if (free-of k2 x)
                  '(* ,(integrate-from-table (exp-op u) (exp-lhs u))
                      ,(unfactorize k2)))))))
```

There are three cases. In any case, all factors are of the form  $(^ u n)$ , so we separate the factor into a base,  $u$ , and exponent,  $n$ . If  $u$  or  $u^n$  evenly divides the original expression (here represented as `factors`), then we have an answer. But we need to check the exponent, because  $\int u^n du$  is  $u^{n+1}/(n+1)$  for  $n \neq -1$ , but it is  $\log(u)$  for  $n = -1$ . But there is a third case to consider. The factor may be something like  $(^ (\sin (^ x 2)) 1)$ , in which case we should consider  $f(u) = \sin(x^2)$ . This case is handled with the help of an integral table. We don't need a derivative table, because we can just use the simplifier for that.

```
(defun deriv (y x) (simplify '(d ,y ,x)))

(defun integration-table (rules)
  (dolist (i-rule rules)
    (let ((rule (infix->prefix i-rule)))
      (setf (get (exp-op (exp-lhs (exp-lhs rule))) 'int)
            rule)))
```

```
(defun in-integral-table? (exp)
  (and (exp-p exp) (get (exp-op exp) 'int)))

(defun integrate-from-table (op arg)
  (let ((rule (get op 'int)))
    (subst arg (exp-lhs (exp-lhs (exp-lhs rule))) (exp-rhs rule)))))

(integration-table
 '((Int log(x) d x = x * log(x) - x)
   (Int exp(x) d x = exp(x))
   (Int sin(x) d x = - cos(x))
   (Int cos(x) d x = sin(x))
   (Int tan(x) d x = - log(cos(x)))
   (Int sinh(x) d x = cosh(x))
   (Int cosh(x) d x = sinh(x))
   (Int tanh(x) d x = log(cosh(x))))
 ))
```

The last step is to install `integrate` as the simplification function for the operator `Int`. The obvious way to do this is:

```
(set-simp-fn 'Int 'integrate)
```

Unfortunately, that does not quite work. The problem is that `integrate` expects two arguments, corresponding to the two arguments  $y$  and  $x$  in  $(\text{Int } y \ x)$ . But the convention for simplification functions is to pass them a single argument, consisting of the whole expression  $(\text{Int } y \ x)$ . We could go back and edit `simplify-exp` to change the convention, but instead I choose to make the conversion this way:

```
(set-simp-fn 'Int #'(lambda (exp)
                      (integrate (exp-lhs exp) (exp-rhs exp))))
```

Here are some examples, taken from chapters 8 and 9 of *Calculus* (Loomis 1974):

```
SIMPLIFIER> (Int x * sin(x ^ 2) d x)
(1/2 * (- (COS (X ^ 2))))
SIMPLIFIER> (Int ((3 * x ^ 3) - 1 / (3 * x ^ 3)) d x)
((3 * ((X ^ 4) / 4)) - (1/3 * ((X ^ -2) / -2)))
SIMPLIFIER> (Int (3 * x + 2) ^ -2/3 d x)
(((3 * X) + 2) ^ 1/3)
SIMPLIFIER> (Int sin(x) ^ 2 * cos(x) d x)
(((SIN X) ^ 3) / 3)
SIMPLIFIER> (Int sin(x) / (1 + cos(x)) d x)
(-1 * (LOG ((COS X) + 1)))
SIMPLIFIER> (Int (2 * x + 1) / (x ^ 2 + x - 1) d x)
```

```
(LOG ((X ^ 2) + (X - 1)))
SIMPLIFIER> (Int 8 * x ^ 2 / (x ^ 3 + 2) ^ 3 d x)
(8 * ((1/3 * (((X ^ 3) + 2) ^ -2)) / -2))
```

All the answers are correct, although the last one could be made simpler. One quick way to simplify such an expression is to factor and unfactor it, and then simplify again:

```
(set-simp-fn 'Int
  #'(lambda (exp)
    (unfactorize
      (factorize
        (integrate (exp-lhs exp) (exp-rhs exp))))))
```

With this change, we get:

```
SIMPLIFIER> (Int 8 * x ^ 2 / (x ^ 3 + 2) ^ 3 d x)
(-4/3 * (((X ^ 3) + 2) ^ -2))
```

## 8.7 History and References

A brief history is given in the introduction to this chapter. An interesting point is that the history of Lisp and of symbolic algebraic manipulation are deeply intertwined. It is not too gross an exaggeration to say that Lisp was invented by John McCarthy to express the symbolic differentiation algorithm. And the development of the first high-quality Lisp system, MacLisp, was driven largely by the needs of MACSYMA, one of the first large Lisp systems. See McCarthy 1958 for early Lisp history and the differentiation algorithm, and Martin and Fateman 1971 and Moses (1975) for more details on MACSYMA. A comprehensive book on computer algebra systems is Davenport 1988. It covers the MACSYMA and REDUCE systems as well as the algorithms behind those systems.

Because symbolic differentiation is historically important, it is presented in a number of text books, from the original Lisp 1.5 Primer (Weissman 1967) and Allen's influential *Anatomy of Lisp* (1978) to recent texts like Brooks 1985, Hennessey 1989, and Tanimoto 1990. Many of these books use rules or data-driven programming, but each treats differentiation as the main task, with simplification as a separate problem. None of them use the approach taken here, where differentiation is just another kind of simplification.

The symbolic integration programs SAINT and SIN are covered in Slagle 1963 and Moses 1967, respectively. The mathematical solution to the problem of integration

in closed term is addressed in Risch 1969, but be warned; this paper is not for the mathematically naive, and it has no hints on programming the algorithm. A better reference is Davenport et al. 1988.

In this book, techniques for improving the efficiency of algebraic manipulation are covered in sections 9.6 and 10.4. Chapter 15 presents a reimplementation that does not use pattern-matching, and is closer to the techniques used in MACSYMA.

## 8.8 Exercises

-  **Exercise 8.2 [s]** Some notations use the operator `**` instead of `^` to indicate exponentiation. Fix `infix->prefix` so that either notation is allowed.
-  **Exercise 8.3 [m]** Can the system as is deal with imaginary numbers? What are some of the difficulties?
-  **Exercise 8.4 [h]** There are some simple expressions involving sums that are not handled by the `integrate` function. The function can integrate  $a \times x^2 + b \times x + c$  but not  $5 \times (a \times x^2 + b \times x + c)$ . Similarly, it can integrate  $x^4 + 2 \times x^3 + x^2$  but not  $(x^2 + x)^2$ , and it can do  $x^3 + x^2 + x + 1$  but not  $(x^2 + 1) \times (x + 1)$ . Modify `integrate` so that it expands out products (or small exponents) of sums. You will probably want to try the usual techniques first, and do the expansion only when that fails.
-  **Exercise 8.5 [d]** Another very general integration technique is called integration by parts. It is based on the rule:

$$\int u dv = uv - \int v du$$

So, for example, given

$$\int x \cos x dx$$

we can take  $u = x, dv = \cos x dx$ . Then we can determine  $v = \sin x$  by integration, and come up with the solution:

$$\int x \cos x dx = x \sin x - \int \sin x \times 1 dx = x \sin x + \cos x$$

It is easy to program an integration by parts routine. The hard part is to program the control component. Integration by parts involves a recursive call to `integrate`, and of all the possible ways of breaking up the original expression into a  $u$  and a  $dv$ ,

few, if any, will lead to a successful integration. One simple control rule is to allow integration by parts only at the top level, not at the recursive level. Implement this approach.

- ?
- Exercise 8.6 [d]** A more complicated approach is to try to decide which ways of breaking up the original expression are promising and which are not. Derive some heuristics for making this division, and reimplement `integrate` to include a search component, using the search tools of chapter 6.

Look in a calculus textbook to see how  $\int \sin^2 x dx$  is evaluated by two integrations by parts and a division. Implement this technique as well.

- ?
- Exercise 8.7 [m]** Write simplification rules for predicate calculus expressions. For example,

```
(true and x = x)
(false and x = false)
(true or x = true)
(false or x = false)
```

- ?
- Exercise 8.8 [m]** The simplification rule ( $x / 0 = \text{undefined}$ ) is necessary to avoid problems with division by zero, but the treatment of `undefined` is inadequate. For example, the expression  $((0 / 0) - (0 / 0))$  will simplify to zero, when it should simplify to `undefined`. Add rules to propagate `undefined` values and prevent them from being simplified away.

- ?
- Exercise 8.9 [d]** Extend the method used to handle `undefined` to handle `+infinity` and `-infinity` as well.

## CHAPTER 9

---

# Efficiency Issues

*A Lisp programmer knows the value of everything,  
but the cost of nothing.*

—Alan J. Perlis

*Lisp is not inherently less efficient than other  
high-level languages.*

—Richard J. Fateman

One of the reasons Lisp has enjoyed a long history is because it is an ideal language for what is now called *rapid-prototyping*—developing a program quickly, with little regards for details. That is what we have done so far in this book: concentrated on getting a working algorithm. Unfortunately, when a prototype is to be turned into a production-quality program, details can no longer be ignored. Most “real” AI programs deal with large amounts of data, and with large search spaces. Thus, efficiency considerations become very important.

However, this does not mean that writing an efficient program is fundamentally different from writing a working program. Ideally, developing an efficient program should be a three-step process. First, develop a working program, using proper abstractions so that the program will be easy to change if necessary. Second, *instrument* the program to determine where it is spending most of the time. Third, replace the slow parts with faster versions, while maintaining the program’s correctness.

The term *efficiency* will be used primarily to talk about the *speed* or run time of a program. To a lesser extent, *efficiency* is also used to refer to the *space* or amount of storage consumed by a program. We will also talk about the *cost* of a program. This is partly a use of the metaphor “time is money,” and partly rooted in actual monetary costs—if a critical program runs unacceptably slowly, you may need to buy a more expensive computer.

Lisp has been saddled with a reputation as an “inefficient language.” Strictly speaking, it makes no sense to call a *language* efficient or inefficient. Rather, it is only a particular *implementation* of the language executing a particular program that can be measured for efficiency. So saying Lisp is inefficient is partly a historical claim: some past implementations *have* been inefficient. It is also partly a prediction: there are some reasons why future implementations are expected to suffer from inefficiencies. These reasons mainly stem from Lisp’s flexibility. Lisp allows many decisions to be delayed until run time, and that can make the run time take longer. In the past decade, the “efficiency gap” between Lisp and “conventional languages” like FORTRAN or C has narrowed. Here are the reasons—some deserved, some not—behind Lisp’s reputation for inefficiency:

- Early implementations were interpreted rather than compiled, which made them inherently inefficient. Common Lisp implementations have compilers, so this is no longer a problem. While Lisp is (primarily) no longer an interpreted language, it is still an *interactive* language, so it retains its flexibility.
- Lisp has often been used to write interpreters for embedded languages, thereby compounding the problem. Consider this quote from Cooper and Wogrin’s (1988) book on the rule-based programming language OPS5:

*The efficiency of implementations that compile rules into executable code compares favorably to that of programs written in most sequential languages such as FORTRAN or Pascal. Implementations that compile rules into data structures to be interpreted, as do many Lisp-based ones, could be noticeably slower.*

Here Lisp is guilty by association. The fallacious chain of reasoning is: Lisp has been used to write interpreters; interpreters are slow; therefore Lisp is slow. While it is true that Lisp makes it very easy to write interpreters, it also makes it easy to write compilers. This book is the first that concentrates on using Lisp as both the implementation and target language for compilers.

- Lisp encourages a style with lots of function calls, particularly recursive calls. In some older systems, function calls were expensive. But it is now understood that a function call can be compiled into a simple branch instruction, and that

many recursive calls can be made no more expensive than an equivalent iterative loop (see chapter 22). It is also possible to instruct a Common Lisp compiler to compile certain functions inline, so there is no calling overhead at all.

On the other hand, many Lisp systems require two fetches instead of one to find the code for a function, and thus will be slower. This extra level of indirection is the price paid for the freedom of being able to redefine functions without reloading the whole program.

- Run-time type-checking is slow. Lisp provides a repertoire of generic functions. For example, we can write `(+ x y)` without bothering to declare if `x` and `y` are integers, floating point, bignums, complex numbers, rationals, or some combination of the above. This is very convenient, but it means that type checks must be made at run time, so the generic `+` will be slower than, say, a 16-bit integer addition with no check for overflow. If efficiency is important, Common Lisp allows the programmer to include declarations that can eliminate run-time checks.

In fact, once the proper declarations are added, Lisp can be as fast or faster than conventional languages. Fateman (1973) compared the FORTRAN cube root routine on the PDP-10 to a MacLisp transliteration. The MacLisp version produced almost identical numerical code, but was 18% faster overall, due to a superior function-calling sequence.<sup>1</sup> The epigraph at the beginning of this chapter is from this article.

Berlin and Weise (1990) show that with a special compilation technique called *partial evaluation*, speeds 7 to 90 times faster than conventionally compiled code can be achieved. Of course, partial evaluation could be used in any language, but it is very easy to do in Lisp.

The fact remains that Lisp objects must somehow represent their type, and even with declarations, not all of this overhead can be eliminated. Most Lisp implementations optimize access to lists and fixnums but pay the price for the other, less commonly used data types.

- Lisp automatically manages storage, and so it must periodically stop and collect the unused storage, or *garbage*. In early systems, this was done by periodically sweeping through all of memory, resulting in an appreciable pause. Modern systems tend to use incremental garbage-collection techniques, so pauses are shorter and usually unnoticed by the user (although the pauses may still be too long for real-time applications such as controlling a laboratory instrument). The problem with automatic garbage collection these days is not that it is slow—in fact, the automatic systems do about as well as handcrafted storage

---

<sup>1</sup>One could say that the FORTRAN compiler was “broken.” This underscores the problem of defining the efficiency of a language—do we judge by the most popular compiler, by the best compiler available, or by the best compiler imaginable?

allocation. The problem is that they make it convenient for the programmer to generate a lot of garbage in the first place. Programmers in conventional languages, who have to clean up their own garbage, tend to be more careful and use static rather than dynamic storage more often. If garbage becomes a problem, the Lisp programmer can just adopt these static techniques.

- Lisp systems are big and leave little room for other programs. Most Lisp systems are designed to be complete environments, within which the programmer does all program development and execution. For this kind of operation, it makes sense to have a large language like Common Lisp with a huge set of tools. However, it is becoming more common to use Lisp as just one component in a computing environment that may include UNIX, X Windows, emacs, and other interacting programs. In this kind of heterogeneous environment, it would be useful to be able to define and run small Lisp processes that do not include megabytes of unused tools. Some recent compilers support this option, but it is not widely available yet.
- Lisp is a complicated high-level language, and it can be difficult for the programmer to anticipate the costs of various operations. In general, the problem is not that an efficient encoding is impossible but that it is difficult to arrive at that efficient encoding. In a language like C, the experienced programmer has a pretty good idea how each statement will compile into assembly language instructions. But in Lisp, very similar statements can compile into widely different assembly-level instructions, depending on subtle interactions between the declarations given and the capabilities of the compiler. Page 318 gives an example where adding a declaration speeds up a trivial function by 40 times. Nonexperts do not understand when such declarations are necessary and are frustrated by the seeming inconsistencies. With experience, the expert Lisp programmer eventually develops a good “efficiency model,” and the need for such declarations becomes obvious. Recent compilers such as CMU’s Python provide feedback that eases this learning process.

In summary, Lisp makes it possible to write programs in a wide variety of styles, some efficient, some less so. The programmer who writes Lisp programs in the same style as C programs will probably find Lisp to be of comparable speed, perhaps slightly slower. The programmer who uses some of the more dynamic features of Lisp typically finds that it is much easier to develop a working program. Then, if the resulting program is not efficient enough, there will be more time to go back and improve critical sections. Deciding which parts of the program use the most resources is called *instrumentation*. It is foolhardy to try to improve the efficiency of a program without first checking if the improvement will make a real difference.

One route to efficiency is to use the Lisp prototype as a specification and reimplement that specification in a lower-level language, such as C or C++. Some commercial

AI vendors are taking this route. An alternative is to use Lisp as the language for both the prototype and the final implementation. By adding declarations and making minor changes to the original program, it is possible to end up with a Lisp program that is similar in efficiency to a C program.

There are four very general and language-independent techniques for speeding up an algorithm:

- *Caching* the results of computations for later reuse.
- *Compiling* so that less work is done at run time.
- *Delaying* the computation of partial results that may never be needed.
- *Indexing* a data structure for quicker retrieval.

This chapter covers each of the four techniques in order. It then addresses the important problem of *instrumentation*. The chapter concludes with a case study of the `simplify` program. The techniques outlined here result in a 130-fold speed-up in this program.

Chapter 10 concentrates on lower-level “tricks” for improving efficiency further.

## 9.1 Caching Results of Previous Computations: Memoization

We start with a simple mathematical function to demonstrate the advantages of caching techniques. Later we will demonstrate more complex examples.

The Fibonacci sequence is defined as the numbers 1, 1, 2, 3, 5, 8, ... where each number is the sum of the two previous numbers. The most straightforward function to compute the nth number in this sequence is as follows:

```
(defun fib (n)
  "Compute the nth number in the Fibonacci sequence."
  (if (<= n 1)
      1
      (+ (fib (- n 1)) (fib (- n 2)))))
```

The problem with this function is that it computes the same thing over and over again. To compute (`fib 5`) means computing (`fib 4`) and (`fib 3`), but (`fib 4`) also requires (`fib 3`), they both require (`fib 2`), and so on. There are ways to rewrite the function to do less computation, but wouldn’t it be nice to write the function as is, and have it automatically avoid redundant computation? Amazingly, there is a way to do just that. The idea is to use the function `fib` to build a new function that remembers previously computed results and uses them, rather than recompute

them. This process is called *memoization*. The function `memo` below is a higher-order function that takes a function as input and returns a new function that will compute the same results, but not do the same computation twice.

```
(defun memo (fn)
  "Return a memo-function of fn."
  (let ((table (make-hash-table)))
    #'(lambda (x)
        (multiple-value-bind (val found-p)
            (gethash x table)
          (if found-p
              val
              (setf (gethash x table) (funcall fn x)))))))
```

The expression `(memo #'fib)` will produce a function that remembers its results between calls, so that, for example, if we apply it to 3 twice, the first call will do the computation of `(fib 3)`, but the second will just look up the result in a hash table. With `fib` traced, it would look like this:

```
> (setf memo-fib (memo #'fib)) => #<CLOSURE -67300731>

> (funcall memo-fib 3) =>
(1 ENTER FIB: 3)
(2 ENTER FIB: 2)
(3 ENTER FIB: 1)
(3 EXIT FIB: 1)
(3 ENTER FIB: 0)
(3 EXIT FIB: 1)
(2 EXIT FIB: 2)
(2 ENTER FIB: 1)
(2 EXIT FIB: 1)
(1 EXIT FIB: 3)
3

> (funcall memo-fib 3) => 3
```

The second time we call `memo-fib` with 3 as the argument, the answer is just retrieved rather than recomputed. But the problem is that during the computation of `(fib 3)`, we still compute `(fib 2)` multiple times. It would be better if even the internal, recursive calls were memoized, but they are calls to `fib`, which is unchanged, not to `memo-fib`. We can solve this problem easily enough with the function `memoize`:

---

```
(defun memoize (fn-name)
  "Replace fn-name's global definition with a memoized version."
  (setf (symbol-function fn-name) (memo (symbol-function fn-name))))
```

When passed a symbol that names a function, `memoize` changes the global definition of the function to a memo-function. Thus, any recursive calls will go first to the memo-function, rather than to the original function. This is just what we want. In the following, we contrast the memoized and unmemoized versions of `fib`. First, a call to `(fib 5)` with `fib` traced:

```
> (fib 5) =>
(1 ENTER FIB: 5)
(2 ENTER FIB: 4)
(3 ENTER FIB: 3)
(4 ENTER FIB: 2)
(5 ENTER FIB: 1)
(5 EXIT FIB: 1)
(5 ENTER FIB: 0)
(5 EXIT FIB: 1)
(4 EXIT FIB: 2)
(4 ENTER FIB: 1)
(4 EXIT FIB: 1)
(3 EXIT FIB: 3)
(3 ENTER FIB: 2)
(4 ENTER FIB: 1)
(4 EXIT FIB: 1)
(4 ENTER FIB: 0)
(4 EXIT FIB: 1)
(3 EXIT FIB: 2)
(2 EXIT FIB: 5)
(2 ENTER FIB: 3)
(3 ENTER FIB: 2)
(4 ENTER FIB: 1)
(4 EXIT FIB: 1)
(4 ENTER FIB: 0)
(4 EXIT FIB: 1)
(3 EXIT FIB: 2)
(3 ENTER FIB: 1)
(3 EXIT FIB: 1)
(2 EXIT FIB: 3)
(1 EXIT FIB: 8)
```

8

We see that `(fib 5)` and `(fib 4)` are each computed once, but `(fib 3)` is computed twice, `(fib 2)` three times, and `(fib 1)` five times. Below we call `(memoize 'fib)` and repeat the calculation. This time, each computation is done only once. Furthermore,

when the computation of (fib 5) is repeated, the answer is returned immediately with no intermediate computation, and a further call to (fib 6) can make use of the value of (fib 5).

```
> (memoize 'fib) => #<CLOSURE 76626607>
> (fib 5) =>
(1 ENTER FIB: 5)
(2 ENTER FIB: 4)
(3 ENTER FIB: 3)
(4 ENTER FIB: 2)
(5 ENTER FIB: 1)
(5 EXIT FIB: 1)
(5 ENTER FIB: 0)
(5 EXIT FIB: 1)
(4 EXIT FIB: 2)
(3 EXIT FIB: 3)
(2 EXIT FIB: 5)
(1 EXIT FIB: 8)
8
> (fib 5) => 8
> (fib 6) =>
(1 ENTER FIB: 6)
(1 EXIT FIB: 13)
13
```

Understanding why this works requires a clear understanding of the distinction between functions and function names. The original (defun fib ...) form does two things: builds a function and stores it as the symbol-function value of fib. Within that function there are two references to fib; these are compiled (or interpreted) as instructions to fetch the symbol-function of fib and apply it to the argument.

What memoize does is fetch the original function and transform it with memo to a function that, when called, will first look in the table to see if the answer is already known. If not, the original function is called, and a new value is placed in the table. The trick is that memoize takes this new function and makes it the symbol-function value of the function name. This means that all the references in the original function will now go to the new function, and the table will be properly checked on each recursive call. One further complication to memo: the function gethash returns both the value found in the table and an indicator of whether the key was present or not. We use multiple-value-bind to capture both values, so that we can distinguish the case when nil is the value of the function stored in the table from the case where there is no stored value.

If you make a change to a memoized function, you need to recompile the original definition, and then redo the call to memoize. In developing your program, rather

than saying (`(memoize 'f)`), it might be easier to wrap appropriate definitions in a `memoize` form as follows:

```
(memoize
  (defun f (x) ...)
  )
```

Or define a macro that combines `defun` and `memoize`:

```
(defmacro defun-memo (fn args &body body)
  "Define a memoized function."
  '(memoize (defun ,fn ,args . ,body)))
(defun-memo f (x) ...)
```

Both of these approaches rely on the fact that `defun` returns the name of the function defined.

<i>n</i>	<code>(fib n)</code>	unmemoized	memoized	memoized up to
25	121393	1.1	.010	0
26	196418	1.8	.001	25
27	317811	2.9	.001	26
28	514229	4.7	.001	27
29	832040	8.2	.001	28
30	1346269	12.4	.001	29
31	2178309	20.1	.001	30
32	3524578	32.4	.001	31
33	5702887	52.5	.001	32
34	9227465	81.5	.001	33
50	2.0e10	—	.014	34
100	5.7e20	—	.031	50
200	4.5e41	—	.096	100
500	2.2e104	—	.270	200
1000	7.0e208	—	.596	500
1000	7.0e208	—	.001	1000
1000	7.0e208	—	.876	0

Now we show a table giving the values of `(fib n)` for certain *n*, and the time in seconds to compute the value, before and after `(memoize 'fib)`. For larger values of *n*, approximations are shown in the table, although `fib` actually returns an exact integer. With the unmemoized version, I stopped at *n* = 34, because the times were getting too long. For the memoized version, even *n* = 1000 took under a second.

Note there are three entries for (fib 1000). The first entry represents the incremental computation when the table contains the memoized values up to 500, the second entry shows the time for a table lookup when (fib 1000) is already computed, and the third entry is the time for a complete computation starting with an empty table.

It should be noted that there are two general approaches to discussing the efficiency of an algorithm. One is to time the algorithm on representative inputs, as we did in this table. The other is to analyze the *asymptotic complexity* of the algorithm. For the fib problem, an asymptotic analysis considers how long it takes to compute (fib  $n$ ) as  $n$  approaches infinity. The notation  $O(f(n))$  is used to describe the complexity. For example, the memoized version fib is an  $O(n)$  algorithm because the computation time is bounded by some constant times  $n$ , for any value of  $n$ . The unmemoized version, it turns out, is  $O(1.7^n)$ , meaning computing fib of  $n+1$  can take up to 1.7 times as long as fib of  $n$ . In simpler terms, the memoized version has *linear* complexity, while the unmemoized version has *exponential* complexity. Exercise 9.4 (page 308) describes where the 1.7 comes from, and gives a tighter bound on the complexity.

The version of memo presented above is inflexible in several ways. First, it only works for functions of one argument. Second, it only returns a stored value for arguments that are eq1, because that is how hash tables work by default. For some applications we want to retrieve the stored value for arguments that are equal. Third, there is no way to delete entries from the hash table. In many applications there are times when it would be good to clear the hash table, either because it has grown too large or because we have finished a set of related problems and are moving on to a new problem.

The versions of memo and memoize below handle these three problems. They are compatible with the previous version but add three new keywords for the extensions. The name keyword stores the hash table on the property list of that name, so it can be accessed by clear-memoize. The test keyword tells what kind of hash table to create: eq, eq1, or equal. Finally, the key keyword tells which arguments of the function to index under. The default is the first argument (to be compatible with the previous version), but any combination of the arguments can be used. If you want to use all the arguments, specify identity as the key. Note that if the key is a list of arguments, then you will have to use equal hash tables.

```
(defun memo (fn name key test)
  "Return a memo-function of fn."
  (let ((table (make-hash-table :test test)))
    (setf (get name 'memo) table)
    #'(lambda (&rest args)
        (let ((k (funcall key args)))
          (multiple-value-bind (val found-p)
              (gethash k table)
            (if found-p val
```

```
(setf (gethash k table) (apply fn args))))))

(defun memoize (fn-name &key (key #'first) (test #'eq))
  "Replace fn-name's global definition with a memoized version."
  (setf (symbol-function fn-name)
        (memo (symbol-function fn-name) fn-name key test)))

(defun clear-memoize (fn-name)
  "Clear the hash table from a memo function."
  (let ((table (get fn-name 'memo)))
    (when table (clrhash table))))
```

## 9.2 Compiling One Language into Another

In chapter 2 we defined a new language—the language of grammar rules—which was processed by an interpreter designed especially for that language. An *interpreter* is a program that looks at some data structure representing a “program” or sequence of rules of some sort and interprets or evaluates those rules. This is in contrast to a *compiler*, which translates some set of rules in one language into a program in another language.

The function `generate` was an interpreter for the “language” defined by the set of grammar rules. Interpreting these rules is straightforward, but the process is somewhat inefficient, in that `generate` must continually search through the `*grammar*` to find the appropriate rule, then count the length of the right-hand side, and so on.

A compiler for this rule-language would take each rule and translate it into a function. These functions could then call each other with no need to search through the `*grammar*`. We implement this approach with the function `compile-rule`. It makes use of the auxiliary functions `one-of` and `rule-lhs` and `rule-rhs` from page 40, repeated here:

```
(defun rule-lhs (rule)
  "The left-hand side of a rule."
  (first rule))

(defun rule-rhs (rule)
  "The right-hand side of a rule."
  (rest (rest rule)))

(defun one-of (set)
  "Pick one element of set, and make a list of it."
  (list (random-elt set)))
```

```
(defun random-elt (choices)
  "Choose an element from a list at random."
  (elt choices (random (length choices))))
```

The function `compile-rule` turns a rule into a function definition by building up Lisp code that implements all the actions that generate would take in interpreting the rule. There are three cases. If every element of the right-hand side is an atom, then the rule is a lexical rule, which compiles into a call to `one-of` to pick a word at random. If there is only one element of the right-hand side, then `build-code` is called to generate code for it. Usually, this will be a call to `append` to build up a list. Finally, if there are several elements in the right-hand side, they are each turned into code by `build-code`; are given a number by `build-cases`; and then a `case` statement is constructed to choose one of the cases.

```
(defun compile-rule (rule)
  "Translate a grammar rule into a LISP function definition."
  (let ((rhs (rule-rhs rule)))
    '(defun ,(rule-lhs rule) ()
       ,(cond ((every #'atom rhs) '(one-of ',rhs))
              ((length=1 rhs) (build-code (first rhs)))
              (t '(case (random ,(length rhs))
                        ,@(build-cases 0 rhs))))))

(defun build-cases (number choices)
  "Return a list of case-clauses"
  (when choices
    (cons (list number (build-code (first choices)))
          (build-cases (+ number 1) (rest choices)))))

(defun build-code (choice)
  "Append together multiple constituents"
  (cond ((null choice) nil)
        ((atom choice) (list choice))
        ((length=1 choice) choice)
        (t '(append ,@(mapcar #'build-code choice)))))

(defun length=1 (x)
  "Is X a list of length 1?"
  (and (consp x) (null (rest x))))
```

The Lisp code built by `compile-rule` must be compiled or interpreted to make it available to the Lisp system. We can do that with one of the following forms. Normally we would want to call `compile`, but during debugging it may be easier not to.

```
(dolist (rule *grammar*) (eval (compile-rule rule)))
(dolist (rule *grammar*) (compile (eval (compile-rule rule))))
```

One frequent way to use compilation is to define a macro that expands into the code generated by the compiler. That way, we just type in calls to the macro and don't have to worry about making sure all the latest rules have been compiled. We might implement this as follows:

```
(defmacro defrule (&rest rule)
  "Define a grammar rule"
  (compile-rule rule))

(defrule Sentence -> (NP VP))
(defrule NP -> (Art Noun))
(defrule VP -> (Verb NP))
(defrule Art -> the a)
(defrule Noun -> man ball woman table)
(defrule Verb -> hit took saw liked)
```

Actually, the choice of using one big list of rules (like `*grammar*`) versus using individual macros to define rules is independent of the choice of compiler versus interpreter. We could just as easily define `defrule` simply to push the rule onto `*grammar*`. Macros like `defrule` are useful when you want to define rules in different places, perhaps in several separate files. The `defparameter` method is appropriate when all the rules can be defined in one place.

We can see the Lisp code generated by `compile-rule` in two ways: by passing it a rule directly:

```
> (compile-rule '(Sentence -> (NP VP)))
(DEFUN SENTENCE ()
  (APPEND (NP) (VP)))

> (compile-rule '(Noun -> man ball woman table))
(DEFUN NOUN ()
  (ONE-OF '(MAN BALL WOMAN TABLE)))
```

or by macroexpanding a `defrule` expression. The compiler was designed to produce the same code we were writing in our first approach to the generation problem (see page 35).

```
> (macroexpand '(defrule Adj* -> () Adj (Adj Adj*)))
(DEFUN ADJ* ()
  (CASE (RANDOM 3)
    (0 NIL)
    (1 (ADJ))
    (2 (APPEND (ADJ) (ADJ*)))))
```

Interpreters are usually easier to write than compilers, although in this case, even the compiler was not too difficult. Interpreters are also inherently more flexible than compilers, because they put off making decisions until the last possible moment. For example, our compiler considers the right-hand side of a rule to be a list of words only if every element is an atom. In all other cases, the elements are treated as nonterminals. This could cause problems if we extended the definition of Noun to include the compound noun “chow chow”:

```
(defrule Noun -> man ball woman table (chow chow))
```

The rule would expand into the following code:

```
(DEFUN NOUN ()
  (CASE (RANDOM 5)
    (0 (MAN))
    (1 (BALL))
    (2 (WOMAN))
    (3 (TABLE))
    (4 (APPEND (CHOW) (CHOW)))))
```

The problem is that `man` and `ball` and all the others are suddenly treated as functions, not as literal words. So we would get a run-time error notifying us of undefined functions. The equivalent rule would cause no trouble for the interpreter, which waits until it actually needs to generate a symbol to decide if it is a word or a nonterminal. Thus, the semantics of rules are different for the interpreter and the compiler, and we as program implementors have to be very careful about how we specify the actual meaning of a rule. In fact, this was probably a bug in the interpreter version, since it effectively prohibits words like “noun” and “sentence” from occurring as words if they are also the names of categories. One possible resolution of the conflict is to say that an element of a right-hand side represents a word if it is an atom, and a list of categories if it is a list. If we did indeed settle on that convention, then we could modify both the interpreter and the compiler to comply with the convention. Another possibility would be to represent words as strings, and categories as symbols.

The flip side of losing run-time flexibility is gaining compile-time diagnostics. For example, it turns out that on the Common Lisp system I am currently using, I get some useful error messages when I try to compile the buggy version of `Noun`:

```
> (defrule Noun -> man ball woman table (chow chow))
The following functions were referenced but don't seem defined:
  CHOW referenced by NOUN
  TABLE referenced by NOUN
  WOMAN referenced by NOUN
  BALL referenced by NOUN
  MAN referenced by NOUN
  NOUN
```

Another problem with the compilation scheme outlined here is the possibility of *name clashes*. Under the interpretation scheme, the only names used were the function generate and the variable \*grammar\*. With compilation, every left-hand side of a rule becomes the name of a function. The grammar writer has to make sure he or she is not using the name of an existing Lisp function, and hence redefining it. Even worse, if more than one grammar is being developed at the same time, they cannot have any functions in common. If they do, the user will have to recompile with every switch from one grammar to another. This may make it difficult to compare grammars. The best away around this problem is to use the Common Lisp idea of *packages*, but for small exercises name clashes can be avoided easily enough, so we will not explore packages until section 24.1.

The major advantage of a compiler is speed of execution, when that makes a difference. For identical grammars running in one particular implementation of Common Lisp on one machine, our interpreter generates about 75 sentences per second, while the compiled approach turns out about 200. Thus, it is more than twice as fast, but the difference is negligible unless we need to generate many thousands of sentences. In section 9.6 we will see another compiler with an even greater speed-up.

The need to optimize the code produced by your macros and compilers ultimately depends on the quality of the underlying Lisp compiler. For example, consider the following code:

```
> (defun f1 (n 1)
  (let ((11 (first 1))
        (12 (second 1)))
    (expt (* 1 (+ n 0))
          (- 4 (length (list 11 12))))))

F1

> (defun f2 (n 1) (* n n)) ⇒ F2

> (disassemble 'f1)
  6 PUSH      ARG10      : N
  7 MOVEM     PDL-PUSH
  8 *
  9 RETURN    PDL-POP
```

```
> (disassemble 'f2)
 6 PUSH      ARGO      ; N
 7 MOVEM    PDL-PUSH
 8 *        PDL-POP
 9 RETURN   PDL-POP
F2
```

This particular Lisp compiler generates the exact same code for f1 and f2. Both functions square the argument n, and the four machine instructions say, “Take the 0th argument, make a copy of it, multiply those two numbers, and return the result.” It’s clear the compiler has some knowledge of the basic Lisp functions. In the case of f1, it was smart enough to get rid of the local variables 11 and 12 (and their initialization), as well as the calls to first, second, length, and list and most of the arithmetic. The compiler could do this because it has knowledge about the functions length and list and the arithmetic functions. Some of this knowledge might be in the form of simplification rules.

As a user of this compiler, there’s no need for me to write clever macros or compilers that generate streamlined code as seen in f2; I can blindly generate code with possible inefficiencies like those in f1, and assume that the Lisp compiler will cover up for my laziness. With another compiler that didn’t know about such optimizations, I would have to be more careful about the code I generate.

### 9.3 Delaying Computation

Back on page 45, we saw a program to generate all strings derivable from a grammar. One drawback of this program was that some grammars produce an infinite number of strings, so the program would not terminate on those grammars.

It turns out that we often want to deal with infinite sets. Of course, we can’t enumerate all the elements of an infinite set, but we should be able to represent the set and pick elements out one at a time. In other words, we want to be able to specify how a set (or other object) is constructed, but delay the actual construction, perhaps doing it incrementally over time. This sounds like a job for closures: we can specify the set constructor as a function, and then call the function some time later. We will implement this approach with the syntax used in Scheme—the macro `delay` builds a closure to be computed later, and the function `force` calls that function and caches away the value. We use structures of type `delay` to implement this. A `delay` structure has two fields: the `value` and the `function`. Initially, the `value` field is undefined, and the `function` field holds the closure that will compute the value. The first time the `delay` is forced, the `function` is called, and its result is stored in the `value` field. The `function` field is then set to nil to indicate that there is no need to call the function again. The function `force` checks if the `function` needs to be called, and returns the

value. If force is passed an argument that is not a delay, it just returns the argument.

```
(defstruct delay (value nil) (function nil))

(defmacro delay (&rest body)
  "A computation that can be executed later by FORCE."
  '(make-delay :function #'(lambda () . ,body)))

(defun force (x)
  "Find the value of x, by computing if it is a delay."
  (if (not (delay-p x))
      x
      (progn
        (when (delay-function x)
          (setf (delay-value x)
                (funcall (delay-function x)))
          (setf (delay-function x) nil))
        (delay-value x))))
```

Here's an example of the use of delay. The list x is constructed using a combination of normal evaluation and delayed evaluation. Thus, the 1 is printed when x is created, but the 2 is not:

```
> (setf x (list (print 1) (delay (print 2)))) =>
1
(1 #S(DELAY :FUNCTION (LAMBDA () (PRINT 2))))
```

The second element is evaluated (and printed) when it is forced. But then forcing it again just retrieves the cached value, rather than calling the function again:

```
> (force (second x)) =>
2
2

> x => (1 #S(DELAY :VALUE 2))

> (force (second x)) => 2
```

Now let's see how delays can be used to build infinite sets. An infinite set will be considered a special case of what we will call a *pipe*: a list with a first component that has been computed, and a rest component that is either a normal list or a delayed value. Pipes have also been called delayed lists, generated lists, and (most commonly) streams. We will use the term *pipe* because *stream* already has a meaning in Common Lisp. The book *Artificial Intelligence Programming* (Charniak et al. 1987)

also calls these structures pipes, reserving streams for delayed structures that do not cache computed results.

To distinguish pipes from lists, we will use the accessors `head` and `tail` instead of `first` and `rest`. We will also use `empty-pipe` instead of `nil`, `make-pipe` instead of `cons`, and `pipe-elt` instead of `elt`. Note that `make-pipe` is a macro that delays evaluation of the tail.

```
(defmacro make-pipe (head tail)
  "Create a pipe by evaluating head and delaying tail."
  '(cons ,head (delay ,tail)))

(defconstant empty-pipe nil)

(defun head (pipe) (first pipe))
(defun tail (pipe) (force (rest pipe)))

(defun pipe-elt (pipe i)
  "The i-th element of a pipe, 0-based"
  (if (= i 0)
      (head pipe)
      (pipe-elt (tail pipe) (- i 1))))
```

Here's a function that can be used to make a large or infinite sequence of integers with delayed evaluation:

```
(defun integers (&optional (start 0) end)
  "A pipe of integers from START to END.
  If END is nil, this is an infinite pipe."
  (if (or (null end) (<= start end))
      (make-pipe start (integers (+ start 1) end))
      nil))
```

And here is an example of its use. The pipe `c` represents the numbers from 0 to infinity. When it is created, only the zeroth element, 0, is evaluated. The computation of the other elements is delayed.

```
> (setf c (integers 0)) => (0 . #S(DELAY :FUNCTION #<CLOSURE -77435477>))
> (pipe-elt c 0) => 0
```

Calling `pipe-elt` to look at the third element causes the first through third elements to be evaluated. The numbers 0 to 3 are cached in the correct positions, and further elements remain unevaluated. Another call to `pipe-elt` with a larger index would force them by evaluating the delayed function.

```

> (pipe-elt c 3) ⇒ 3

> c ⇒
(0 . #S(DELAY
  :VALUE
  (1 . #S(DELAY
    :VALUE
    (2 . #S(DELAY
      :VALUE
      (3 . #S(DELAY
        :FUNCTION
        #<CLOSURE -77432724>)))))))

```

While this seems to work fine, there is a heavy price to pay. Every delayed value must be stored in a two-element structure, where one of the elements is a closure. Thus, there is some storage wasted. There is also some time wasted, as `tail` or `pipe-elt` must traverse the structures.

An alternate representation for pipes is as  $(value . closure)$  pairs, where the closure values are stored into the actual cons cells as they are computed. Previously we needed structures of type `delay` to distinguish a delayed from a nondelayed object, but in a pipe we know the rest can be only one of three things: nil, a list, or a delayed value. Thus, we can use the closures directly instead of using `delay` structures, if we have some way of distinguishing closures from lists. Compiled closures are atoms, so they can always be distinguished from lists. But sometimes closures are implemented as lists beginning with `lambda` or some other implementation-dependent symbol.<sup>2</sup> The built-in function `functionp` is defined to be true of such lists, as well as of all symbols and all objects returned by `compile`. But using `functionp` means that we can not have a pipe that includes the symbol `lambda` as an element, because it will be confused for a closure:

```
> (functionp (last '(theta iota kappa lambda))) ⇒ T
```

If we consistently use compiled functions, then we could eliminate the problem by testing with the built-in predicate `compiled-function-p`. The following definitions do not make this assumption:

```
(defmacro make-pipe (head tail)
  "Create a pipe by evaluating head and delaying tail."
  '(cons ,head #'(lambda () ,tail)))
```

---

<sup>2</sup>In KCL, the symbol `lambda-closure` is used, and in Allegro, it is `excl::lexical-closure`.

```
(defun tail (pipe)
  "Return tail of pipe or list, and destructively update
  the tail if it is a function."
  (if (functionp (rest pipe))
      (setf (rest pipe) (funcall (rest pipe)))
      (rest pipe)))
```

Everything else remains the same. If we recompile `integers` (because it uses the macro `make-pipe`), we see the following behavior. First, creation of the infinite pipe `c` is similar:

```
> (setf c (integers 0)) => (0 . #<CLOSURE 77350123>)
> (pipe-elt c 0) => 0
```

Accessing an element of the pipe forces evaluation of all the intervening elements, and as before leaves subsequent elements unevaluated:

```
> (pipe-elt c 5) => 5
> c => (0 1 2 3 4 5 . #<CLOSURE 77351636>)
```

Pipes can also be used for finite lists. Here we see a pipe of length 11:

```
> (setf i (integers 0 10)) => (0 . #<CLOSURE 77375357>)
> (pipe-elt i 10) => 10
> (pipe-elt i 11) => NIL
> i => (0 1 2 3 4 5 6 7 8 9 10)
```

Clearly, this version wastes less space and is much neater about cleaning up after itself. In fact, a completely evaluated pipe turns itself into a list! This efficiency was gained at the sacrifice of a general principle of program design. Usually we strive to build more complicated abstractions, like pipes, out of simpler ones, like delays. But in this case, part of the functionality that delays were providing was duplicated by the cons cells that make up pipes, so the more efficient implementation of pipes does not use delays at all.

Here are some more utility functions on pipes:

```
(defun enumerate (pipe &key count key (result pipe))
  "Go through all (or count) elements of pipe,
  possibly applying the KEY function. (Try PRINT.)"
  ;;= Returns RESULT, which defaults to the pipe itself.
  (if (or (eq pipe empty-pipe) (eql count 0))
```

```

result
(progn
  (unless (null key) (funcall key (head pipe)))
  (enumerate (tail pipe) :count (if count (- count 1))
             :key key :result result)))

(defun filter (pred pipe)
  "Keep only items in pipe satisfying pred."
  (if (funcall pred (head pipe))
      (make-pipe (head pipe)
                 (filter pred (tail pipe)))
      (filter pred (tail pipe))))

```

And here's an application of pipes: generating prime numbers using the sieve of Eratosthenes algorithm:

```

(defun sieve (pipe)
  (make-pipe (head pipe)
             (filter #'(lambda (x) (/= (mod x (head pipe)) 0))
                     (sieve (tail pipe)))))

(defvar *primes* (sieve (integers 2)))

> *primes* => (2 . #<CLOSURE 3075345>)

> (enumerate *primes* :count 10) =>
(2 3 5 7 11 13 17 19 23 29 31 . #<CLOSURE 5224472>)

```

Finally, let's return to the problem of generating all strings in a grammar. First we're going to need some more utility functions:

```

(defun map-pipe (fn pipe)
  "Map fn over pipe, delaying all but the first fn call."
  (if (eq pipe empty-pipe)
      empty-pipe
      (make-pipe (funcall fn (head pipe))
                 (map-pipe fn (tail pipe)))))

(defun append-pipes (x y)
  "Return a pipe that appends the elements of x and y."
  (if (eq x empty-pipe)
      y
      (make-pipe (head x)
                 (append-pipes (tail x) y))))

```

```
(defun mappend-pipe (fn pipe)
  "Lazily map fn over pipe, appending results."
  (if (eq pipe empty-pipe)
      empty-pipe
      (let ((x (funcall fn (head pipe))))
        (make-pipe (head x)
                   (append-pipes (tail x)
                                 (mappend-pipe
                                   fn (tail pipe)))))))
```

Now we can rewrite `generate-all` and `combine-all` to use pipes instead of lists. Everything else is the same as on page 45.

```
(defun generate-all (phrase)
  "Generate a random sentence or phrase"
  (if (listp phrase)
      (if (null phrase)
          (list nil)
          (combine-all-pipes
            (generate-all (first phrase))
            (generate-all (rest phrase))))
      (let ((choices (rule-rhs (assoc phrase *grammar*))))
        (if choices
            (mappend-pipe #'generate-all choices)
            (list (list phrase)))))

(defun combine-all-pipes (xpipe ypipe)
  "Return a pipe of pipes formed by appending a y to an x"
  ;;= In other words, form the cartesian product.
  (mappend-pipe
    #'(lambda (y)
        (map-pipe #'(lambda (x) (append-pipes x y))
                  xpipe))
    ypipe))
```

With these definitions, here's the pipe of all sentences from `*grammar2*` (from page 43):

```
> (setf ss (generate-all 'sentence)) =>
((THE . #<CLOSURE 27265720>) . #<CLOSURE 27266035>)
```

```

> (enumerate ss :count 5) ⇒
((THE . #<CLOSURE 27265720>)
(A . #<CLOSURE 27273143>)
(THE . #<CLOSURE 27402545>)
(A . #<CLOSURE 27404344>)
(THE . #<CLOSURE 27404527>)
(A . #<CLOSURE 27405473>) . #<CLOSURE 27405600>

> (enumerate ss :count 5 :key #'enumerate) ⇒
((THE MAN HIT THE MAN)
(A MAN HIT THE MAN)
(THE BIG MAN HIT THE MAN)
(A BIG MAN HIT THE MAN)
(THE LITTLE MAN HIT THE MAN)
(THE . #<CLOSURE 27423236>) . #<CLOSURE 27423343>

> (enumerate (pipe-elt ss 200)) ⇒
(THE ADIABATIC GREEN BLUE MAN HIT THE MAN)

```

While we were able to represent the infinite set of sentences and enumerate instances of it, we still haven't solved all the problems. For one, this enumeration will never get to a sentence that does not have "hit the man" as the verb phrase. We will see longer and longer lists of adjectives, but no other change. Another problem is that left-recursive rules will still cause infinite loops. For example, if the expansion for  $\text{Adj}^*$  had been  $(\text{Adj}^* \rightarrow (\text{Adj}^* \text{ Adj}) ())$  instead of  $(\text{Adj}^* \rightarrow () (\text{Adj} \text{ Adj}^*))$ , then the enumeration would never terminate, because pipes need to generate a first element.

We have used delays and pipes for two main purposes: to put off until later computations that may not be needed at all, and to have an explicit representation of large or infinite sets. It should be mentioned that the language Prolog has a different solution to the first problem (but not the second). As we shall see in chapter 11, Prolog generates solutions one at a time, automatically keeping track of possible backtrack points. Where pipes allow us to represent an infinite number of alternatives in the data, Prolog allows us to represent those alternatives in the program itself.

-  **Exercise 9.1 [h]** When given a function  $f$  and a pipe  $p$ ,  $\text{mappend-pipe}$  returns a new pipe that will eventually enumerate all of  $(f \text{ (first } p))$ , then all of  $(f \text{ (second } p))$ , and so on. This is deemed "unfair" if  $(f \text{ (first } p))$  has an infinite number of elements. Define a function that will fairly interleave elements, so that all of them are eventually enumerated. Show that the function works by changing  $\text{generate-all}$  to work with it.

## 9.4 Indexing Data

Lisp makes it very easy to use lists as the universal data structure. A list can represent a set or an ordered sequence, and a list with sublists can represent a tree or graph. For rapid prototyping, it is often easiest to represent data in lists, but for efficiency this is not always the best idea. To find an element in a list of length  $n$  will take  $n/2$  steps on average. This is true for a simple list, an association list, or a property list. If  $n$  can be large, it is worth looking at other data structures, such as hash tables, vectors, property lists, and trees.

Picking the right data structure and algorithm is as important in Lisp as it is in any other programming language. Even though Lisp offers a wide variety of data structures, it is often worthwhile to spend some effort on building just the right data structure for frequently used data. For example, Lisp's hash tables are very general and thus can be inefficient. You may want to build your own hash tables if, for example, you never need to delete elements, thus making open hashing an attractive possibility. We will see an example of efficient indexing in section 9.6 (page 297).

## 9.5 Instrumentation: Deciding What to Optimize

Because Lisp is such a good rapid-prototyping language, we can expect to get a working implementation quickly. Before we go about trying to improve the efficiency of the implementation, it is a good idea to see what parts are used most often. Improving little-used features is a waste of time.

The minimal support we need is to count the number of calls to selected functions, and then print out the totals. This is called *profiling* the functions.<sup>3</sup> For each function to be profiled, we change the definition so that it increments a counter and then calls the original function.

Most Lisp systems have some built-in profiling mechanism. If your system has one, by all means use it. The code in this section is provided for those who lack such a feature, and as an example of how functions can be manipulated. The following is a simple profiling facility. For each profiled function, it keeps a count of the number of times it is called under the `profile-count` property of the function's name.

---

<sup>3</sup>The terms *metering* and *monitoring* are sometimes used instead of profiling.

```

(defun profile1 (fn-name)
  "Make the function count how often it is called"
  ;; First save away the old, unprofiled function
  ;; Then make the name be a new function that increments
  ;; a counter and then calls the original function
  (let ((fn (symbol-function fn-name)))
    (setf (get fn-name 'unprofiled-fn) fn)
    (setf (get fn-name 'profile-count) 0)
    (setf (symbol-function fn-name)
          (profiled-fn fn-name fn))
    fn-name))

(defun unprofile1 (fn-name)
  "Make the function stop counting how often it is called."
  (setf (symbol-function fn-name) (get fn-name 'unprofiled-fn))
  fn-name)

(defun profiled-fn (fn-name fn)
  "Return a function that increments the count."
  #'(lambda (&rest args)
      (incf (get fn-name 'profile-count))
      (apply fn args)))

(defun profile-count (fn-name) (get fn-name 'profile-count))

(defun profile-report (fn-names &optional (key #'profile-count))
  "Report profiling statistics on given functions."
  (loop for name in (sort fn-names #> :key key) do
        (format t "~&~7D ~A" (profile-count name) name)))

```

That's all we need for the bare-bones functionality. However, there are a few ways we could improve this. First, it would be nice to have macros that, like trace and untrace, allow the user to profile multiple functions at once and keep track of what has been profiled. Second, it can be helpful to see the length of time spent in each function, as well as the number of calls.

Also, it is important to avoid profiling a function twice, since that would double the number of calls reported without alerting the user of any trouble. Suppose we entered the following sequence of commands:

```

(defun f (x) (g x))
(profile1 'f)
(profile1 'f)

```

Then the definition of `f` would be roughly:

```
(lambda (&rest args)
  (incf (get 'f 'profile-count))
  (apply #'(lambda (&rest args)
              (incf (get 'f 'profile-count))
              (apply #'(lambda (x) (g x))
                     args))
         args))
```

The result is that any call to *f* will eventually call the original *f*, but only after incrementing the count twice.

Another consideration is what happens when a profiled function is redefined by the user. The only way we could ensure that a redefined function would continue profiling would be to change the definition of the macro defun to look for functions that should be profiled. Changing system functions like defun is a risky prospect, and in *Common Lisp the Language*, 2d edition, it is explicitly disallowed. Instead, we'll do the next best thing: ensure that the next call to profile will reprofile any functions that have been redefined. We do this by keeping track of both the original unprofiled function and the profiled function. We also keep a list of all functions that are currently profiled.

In addition, we will count the amount of time spent in each function. However, the user is cautioned not to trust the timing figures too much. First, they include the overhead cost of the profiling facility. This can be significant, particularly because the facility conses, and thus can force garbage collections that would not otherwise have been done. Second, the resolution of the system clock may not be fine enough to make accurate timings. For functions that take about 1/10 of a second or more, the figures will be reliable, but for quick functions they may not be.

Here is the basic code for profile and unprofile:

```
(defvar *profiled-functions* nil
  "Function names that are currently profiled")

(defmacro profile (&rest fn-names)
  "Profile fn-names. With no args, list profiled functions."
  '(mapcar #'profile1
            (setf *profiled-functions*
                  (union *profiled-functions* ',fn-names)))))

(defmacro unprofile (&rest fn-names)
  "Stop profiling fn-names. With no args, stop all profiling."
  '(progn
      (mapcar #'unprofile1
              ,(if fn-names '',fn-names '*profiled-functions*)))
      (setf *profiled-functions*
            ,(if (null fn-names)
                 nil)))
```

---

```
(set-difference *profiled-functions*
  ',fn-names))))
```

The idiom `' ,fn-names` deserves comment, since it is common but can be confusing at first. It may be easier to understand when written in the equivalent form `'(quote ,fn-names)`. As always, the backquote builds a structure with both constant and evaluated components. In this case, the quote is constant and the variable `fn-names` is evaluated. In MacLisp, the function `kwote` was defined to serve this purpose:

```
(defun kwote (x) (list 'quote x))
```

Now we need to change `profile1` and `unprofile1` to do the additional bookkeeping: For `profile1`, there are two cases. If the user does a `profile1` on the same function name twice in a row, then on the second time we will notice that the current function is the same as the functioned stored under the `profiled-fn` property, so nothing more needs to be done. Otherwise, we create the profiled function, store it as the current definition of the name under the `profiled-fn` property, save the unprofiled function, and initialize the counts.

```
(defun profile1 (fn-name)
  "Make the function count how often it is called"
  ;; First save away the old, unprofiled function
  ;; Then make the name be a new function that increments
  ;; a counter and then calls the original function
  (let ((fn (symbol-function fn-name)))
    (unless (eq fn (get fn-name 'profiled-fn))
      (let ((new-fn (profiled-fn fn-name fn)))
        (setf (symbol-function fn-name) new-fn
              (get fn-name 'profiled-fn) new-fn
              (get fn-name 'unprofiled-fn) fn
              (get fn-name 'profile-time) 0
              (get fn-name 'profile-count) 0)))
    fn-name)

  (defun unprofile1 (fn-name)
    "Make the function stop counting how often it is called."
    (setf (get fn-name 'profile-time) 0)
    (setf (get fn-name 'profile-count) 0)
    (when (eq (symbol-function fn-name) (get fn-name 'profiled-fn))
      ;; normal case: restore unprofiled version
      (setf (symbol-function fn-name)
            (get fn-name 'unprofiled-fn)))
    fn-name))
```

Now we look into the question of timing. There is a built-in Common Lisp function, `get-internal-real-time`, that returns the elapsed time since the Lisp session started. Because this can quickly become a bignum, some implementations provide another timing function that wraps around rather than increasing forever, but which may have a higher resolution than `get-internal-real-time`. For example, on TI Explorer Lisp Machines, `get-internal-real-time` measures 1/60-second intervals, while `time:microsecond-time` measures 1/1,000,000-second intervals, but the value returned wraps around to zero every hour or so. The function `time:microsecond-time-difference` is used to compare two of these numbers with compensation for wraparound, as long as no more than one wraparound has occurred.

In the code below, I use the conditional read macro characters `#+` and `#-` to define the right behavior on both Explorer and non-Explorer machines. We have seen that `#` is a special character to the reader that takes different action depending on the following character. For example, `#'fn` is read as `(function fn)`. The character sequence `#+` is defined so that `#+feature expression` reads as *expression* if the *feature* is defined in the current implementation, and as nothing at all if it is not. The sequence `#-` acts in just the opposite way. For example, on a TI Explorer, we would get the following:

```
> '(hi #+TI t #+Symbolics s #-Explorer e #-Mac m) => (HI T M)
```

The conditional read macro characters are used in the following definitions:

```
(defun get-fast-time ()
  "Return the elapsed time. This may wrap around;
  use FAST-TIME-DIFFERENCE to compare."
  #+Explorer (time:microsecond-time) ; do this on an Explorer
  #-Explorer (get-internal-real-time)) ; do this on a non-Explorer

(defun fast-time-difference (end start)
  "Subtract two time points."
  #+Explorer (time:microsecond-time-difference end start)
  #-Explorer (- end start))

(defun fast-time->seconds (time)
  "Convert a fast-time interval into seconds."
  #+Explorer (/ time 1000000.0)
  #-Explorer (/ time internal-time-units-per-second))
```

The next step is to update `profiled-fn` to keep track of the timing data. The simplest way to do this would be to set a variable, say `start`, to the time when a function is entered, run the function, and then increment the function's time by the difference between the current time and `start`. The problem with this approach is that every func-

tion in the call stack gets credit for the time of each called function. Suppose the function `f` calls itself recursively five times, with each call and return taking place a second apart, so that the whole computation takes nine seconds. Then `f` will be charged nine seconds for the outer call, seven seconds for the next call, and so on, for a total of 25 seconds, even though in reality it only took nine seconds for all of them together.

A better algorithm would be to charge each function only for the time since the last call or return. Then `f` would only be charged the nine seconds. The variable `*profile-call-stack*` is used to hold a stack of function name/entry time pairs. This stack is manipulated by `profile-enter` and `profile-exit` to get the right timings.

The functions that are used on each call to a profiled function are declared `inline`. In most cases, a call to a function compiles into machine instructions that set up the argument list and branch to the location of the function's definition. With an `inline` function, the body of the function is compiled in line at the place of the function call. Thus, there is no overhead for setting up the argument list and branching to the definition. An `inline` declaration can appear anywhere any other declaration can appear. In this case, the function `proclaim` is used to register a global declaration. Inline declarations are discussed in more depth on page 317.

```
(proclaim '(inline profile-enter profile-exit inc-profile-time))

(defun profiled-fn (fn-name fn)
  "Return a function that increments the count, and times."
  #'(lambda (&rest args)
      (profile-enter fn-name)
      (multiple-value-prog1
        (apply fn args)
        (profile-exit fn-name)))))

(defvar *profile-call-stack* nil)

(defun profile-enter (fn-name)
  (incf (get fn-name 'profile-count))
  (unless (null *profile-call-stack*)
    ; Time charged against the calling function:
    (inc-profile-time (first *profile-call-stack*)
                      (car (first *profile-call-stack*)))))
  ; Put a new entry on the stack
  (push (cons fn-name (get-fast-time))
        *profile-call-stack*))

(defun profile-exit (fn-name)
  ; Time charged against the current function:
  (inc-profile-time (pop *profile-call-stack*)
                    fn-name)
  ; Change the top entry to reflect current time
  (unless (null *profile-call-stack*)
    (setf (cdr (first *profile-call-stack*))
          (get-fast-time))))
```

```
(defun inc-profile-time (entry fn-name)
  (incf (get fn-name 'profile-time)
        (fast-time-difference (get-fast-time) (cdr entry))))
```

Finally, we need to update profile-report to print the timing data as well as the counts. Note that the default fn-names is a copy of the global list. That is because we pass fn-names to sort, which is a destructive function. We don't want the global list to be modified as a result of this sort.

```
(defun profile-report (&optional
                        (fn-names (copy-list *profiled-functions*))
                        (key #'profile-count))
  "Report profiling statistics on given functions."
  (let ((total-time (reduce #'+ (mapcar #'profile-time fn-names))))
    (unless (null key)
      (setf fn-names (sort fn-names #'> :key key)))
    (format t "~&Total elapsed time: ~d seconds."
            (fast-time->seconds total-time))
    (format t "& Count Secs Time% Name")
    (loop for name in fn-names do
          (format t "~&~7D ~6,2F ~3d% ~A"
                  (profile-count name)
                  (fast-time->seconds (profile-time name))
                  (round (/ (profile-time name) total-time) .01)
                  name)))
  (defun profile-time (fn-name) (get fn-name 'profile-time)))
```

These functions can be used by calling profile, then doing some representative computation, then calling profile-report, and finally unprofile. It can be convenient to provide a single macro for doing all of these at once:

```
(defmacro with-profiling (fn-names &rest body)
  '(progn
    (unprofile . ,fn-names)
    (profile . ,fn-names)
    (setf *profile-call-stack* nil)
    (unwind-protect
      (progn . ,body)
      (profile-report ',fn-names)
      (unprofile . ,fn-names))))
```

Note the use of unwind-protect to produce the report and call unprofile even if the computation is aborted. unwind-protect is a special form that takes any number of arguments. It evaluates the first argument, and if all goes well it then evaluates

the other arguments and returns the first one, just like `prog1`. But if an error occurs during the evaluation of the first argument and computation is aborted, then the subsequent arguments (called cleanup forms) are evaluated anyway.

## 9.6 A Case Study in Efficiency: The SIMPLIFY Program

Suppose we wanted to speed up the `simplify` program of chapter 8. This section shows how a combination of general techniques—memoizing, indexing, and compiling—can be used to speed up the program by a factor of 130. Chapter 15 will show another approach: replace the algorithm with an entirely different one.

The first step to a faster program is defining a *benchmark*, a test suite representing a typical work load. The following is a short list of test problems (and their answers) that are typical of the `simplify` task.

```
(defvar *test-data* (mapcar #'infix->prefix
  '((d (a * x ^ 2 + b * x + c) / d x)
    (d ((a * x ^ 2 + b * x + c) / x) / d x)
    (d ((a * x ^ 3 + b * x ^ 2 + c * x + d) / x ^ 5) / d x)
    ((sin (x + x)) * (sin (2 * x)) + (cos (d (x ^ 2) / d x)) ^ 1)
    (d (3 * x + (cos x) / x) / d x))))
(defvar *answers* (mapcar #'simplify *test-data*))
```

The function `test-it` runs through the test data, making sure that each answer is correct and optionally printing profiling data.

```
(defun test-it (&optional (with-profiling t))
  "Time a test run, and make sure the answers are correct."
  (let ((answers
         (if with-profiling
             (with-profiling (simplify simplify-exp pat-match
                                         match-variable variable-p)
                           (mapcar #'simplify *test-data*))
             (time (mapcar #'simplify *test-data*))))
        (mapc #'assert-equal answers *answers*)))
    t))

(defun assert-equal (x y)
  "If x is not equal to y, complain."
  (assert (equal x y) (x y)
          "Expected ~a to be equal to ~a" x y))
```

Here are the results of (`test-it`) with and without profiling:

```

> (test-it nil)
Evaluation of (MAPCAR #'SIMPLIFY *TEST-DATA*) took 6.612 seconds.

> (test-it t)
Total elapsed time: 22.819614 seconds.
  Count  Secs Time% Name
  51690  11.57  51% PAT-MATCH
  37908   8.75  38% VARIABLE-P
   1393   0.32   1% MATCH-VARIABLE
    906   0.20   1% SIMPLIFY
    274   1.98   9% SIMPLIFY-EXP

```

Running the test takes 6.6 seconds normally, although the time triples when the profiling overhead is added in. It should be clear that to speed things up, we have to either speed up or cut down on the number of calls to pat-match or variable-p, since together they account for 89% of the calls (and 89% of the time as well). We will look at three methods for achieving both those goals.

### *Memoization*

Consider the rule that transforms  $(x + x)$  into  $(2 * x)$ . Once this is done, we have to simplify the result, which involves resimplifying the components. If  $x$  were some complex expression, this could be time-consuming, and it will certainly be wasteful, because  $x$  is already simplified and cannot change. We have seen this type of problem before, and the solution is memoization: make simplify remember the work it has done, rather than repeating the work. We can just say:

```
(memoize 'simplify :test #'equal)
```

Two questions are unclear: what kind of hash table to use, and whether we should clear the hash table between problems. The simplifier was timed for all four combinations of eq or equal hash tables and resetting or nonresetting between problems. The fastest result was equal hashing and nonresetting. Note that with eq hashing, the resetting version was faster, presumably because it couldn't take advantage of the common subexpressions between examples (since they aren't eq).

hashing	resetting	time
none	—	6.6
equal	yes	3.8
equal	no	3.0
eq	yes	7.0
eq	no	10.2

This approach makes the function `simplify` remember the work it has done, in a hash table. If the overhead of hash table maintenance becomes too large, there is an alternative: make the data remember what `simplify` has done. This approach was taken in MACSYMA: it represented operators as lists rather than as atoms. Thus, instead of  $(\star 2 x)$ , MACSYMA would use  $((\star) 2 x)$ . The simplification function would destructively insert a marker into the operator list. Thus, the result of simplifying  $2x$  would be  $((\star \text{simp}) 2 x)$ . Then, when the simplifier was called recursively on this expression, it would notice the `simp` marker and return the expression as is.

The idea of associating memoization information with the data instead of with the function will be more efficient unless there are many functions that all want to place their marks on the same data. The data-oriented approach has two drawbacks: it doesn't identify structures that are `equal` but not `eq`, and, because it requires explicitly altering the data, it requires every other operation that manipulates the data to know about the markers. The beauty of the hash table approach is that it is transparent; no code needs to know that memoization is taking place.

### *Indexing*

We currently go through the entire list of rules one at a time, checking each rule. This is inefficient because most of the rules could be trivially ruled out—if only they were indexed properly. The simplest indexing scheme would be to have a separate list of rules indexed under each operator. Instead of having `simplify-exp` check each member of `*simplification-rules*`, it could look only at the smaller list of rules for the appropriate operator. Here's how:

```
(defun simplify-exp (exp)
  "Simplify using a rule, or by doing arithmetic,
  or by using the simp function supplied for this operator.
  This version indexes simplification rules under the operator."
  (cond ((simplify-by-fn exp)
         ((rule-based-translator exp (rules-for (exp-op exp)) ;***
           :rule-if #'exp-lhs :rule-then #'exp-rhs
           :action #'(lambda (bindings response)
                       (simplify (sublis bindings response)))))

         ((evaluable exp) (eval exp))
         (t exp)))

  (defvar *rules-for* (make-hash-table :test #'eq))

  (defun main-op (rule) (exp-op (exp-lhs rule))))
```

```
(defun index-rules (rules)
  "Index all the rules under the main op."
  (clrhash *rules-for*)
  (dolist (rule rules)
    ;;= nconc instead of push to preserve the order of rules
    (setf (gethash (main-op rule) *rules-for*)
          (nconc (gethash (main-op rule) *rules-for*)
                 (list rule)))))

(defun rules-for (op) (gethash op *rules-for*))

(index-rules *simplification-rules*)
```

Timing the memoized, indexed version gets us to .98 seconds, down from 6.6 seconds for the original code and 3 seconds for the memoized code. If this hadn't helped, we could have considered more sophisticated indexing schemes. Instead, we move on to consider other means of gaining efficiency.

-  **Exercise 9.2 [m]** The list of rules for each operator is stored in a hash table with the operator as key. An alternative would be to store the rules on the property list of each operator, assuming operators must be symbols. Implement this alternative, and time it against the hash table approach. Remember that you need some way of clearing the old rules—trivial with a hash table, but not automatic with property lists.

### *Compilation*

You can look at `simplify-exp` as an interpreter for the simplification rule language. One proven technique for improving efficiency is to replace the interpreter with a compiler. For example, the rule  $(x + x = 2 * x)$  could be compiled into something like:

```
(lambda (exp)
  (if (and (eq (exp-op exp) '+) (equal (exp-lhs exp) (exp-rhs exp)))
       (make-exp :op '* :lhs 2 :rhs (exp-rhs exp))))
```

This eliminates the need for consing up and passing around variable bindings, and should be faster than the general matching procedure. When used in conjunction with indexing, the individual rules can be simpler, because we already know we have the right operator. For example, with the above rule indexed under "+", it could now be compiled as:

---

```
(lambda (exp)
  (if (equal (exp-lhs exp) (exp-rhs exp))
      (make-exp :op '* :lhs 2 :rhs (exp-lhs exp))))
```

It is important to note that when these functions return nil, it means that they have failed to simplify the expression, and we have to consider another means of simplification.

Another possibility is to compile a set of rules all at the same time, so that the indexing is in effect part of the compiled code. As an example, I show here a small set of rules and a possible compilation of the rule set. The generated function assumes that  $x$  is not an atom. This is appropriate because we are replacing `simplify-exp`, not `simplify`. Also, we will return nil to indicate that  $x$  is already simplified. I have chosen a slightly different format for the code; the main difference is the `let` to introduce variable names for subexpressions. This is useful especially for deeply nested patterns. The other difference is that I explicitly build up the answer with a call to `list`, rather than `make-exp`. This is normally considered bad style, but since this is code generated by a compiler, I wanted it to be as efficient as possible. If the representation of the `exp` data type changed, we could simply change the compiler; a much easier task than hunting down all the references spread throughout a human-written program. The comments following were not generated by the compiler.

```
(x * 1 = x)
(1 * x = x)
(x * 0 = 0)
(0 * x = 0)
(x * x = x ^ 2)

(lambda (x)
  (let ((x1 (exp-lhs x))
        (xr (exp-rhs x)))
    (or (if (eql xr '1)           ; (x * 1 = x)
           x1)
        (if (eql x1 '1)           ; (1 * x = x)
            xr)
        (if (eql xr '0)           ; (x * 0 = 0)
            '0)
        (if (eql x1 '0)           ; (0 * x = 0)
            '0)
        (if (equal xr x1)         ; (x * x = x ^ 2)
            (list '^ x1 '2))))))
```

I chose this format for the code because I imagined (and later showed) that it would be fairly easy to write the compiler for it.

### *The Single-Rule Compiler*

Here I show the complete single-rule compiler, to be followed by the indexed-rule-set compiler. The single-rule compiler works like this:

```
> (compile-rule '(= (+ x x) (* 2 x)))
  (LAMBDA (X)
    (IF (OP? X '+)
        (LET ((XL (EXP-LHS X))
              (XR (EXP-RHS X)))
          (IF (EQUAL XR XL)
              (SIMPLIFY-EXP (LIST '* '2 XL))))))
```

Given a rule, it generates code that first tests the pattern and then builds the right-hand side of the rule if the pattern matches. As the code is generated, correspondences are built between variables in the pattern, like *x*, and variables in the generated code, like *x1*. These are kept in the association list *\*bindings\**. The matching can be broken down into four cases: variables that haven't been seen before, variables that have been seen before, atoms, and lists. For example, the first time we run across *x* in the rule above, no test is generated, since anything can match *x*. But the entry (*x . x1*) is added to the *\*bindings\** list to mark the equivalence. When the second *x* is encountered, the test (*equal xr x1*) is generated.

Organizing the compiler is a little tricky, because we have to do three things at once: return the generated code, keep track of the *\*bindings\**, and keep track of what to do "next"—that is, when a test succeeds, we need to generate more code, either to test further, or to build the result. This code needs to know about the bindings, so it can't be done *before* the first part of the test, but it also needs to know where it should be placed in the overall code, so it would be messy to do it *after* the first part of the test. The answer is to pass in a function that will tell us what code to generate later. This way, it gets done at the right time, and ends up in the right place as well. Such a function is often called a *continuation*, because it tells us where to continue computing. In our compiler, the variable consequent is a continuation function.

The compiler is called *compile-rule*. It takes a rule as an argument and returns a lambda expression that implements the rule.

```
(defvar *bindings* nil
  "A list of bindings used by the rule compiler.")

(defun compile-rule (rule)
  "Compile a single rule."
  (let ((*bindings* nil))
    `(lambda (x)
       ,(compile-exp 'x (exp-lhs rule)) ; x is the lambda parameter
       (delay (build-exp (exp-rhs rule))))))
```

```
*bindings*))))))
```

All the work is done by `compile-exp`, which takes three arguments: a variable that will represent the input in the generated code, a pattern that the input should be matched against, and a continuation for generating the code if the test passes. There are five cases: (1) If the pattern is a variable in the list of bindings, then we generate an equality test. (2) If the pattern is a variable that we have not seen before, then we add it to the binding list, generate no test (because anything matches a variable) and then generate the consequent code. (3) If the pattern is an atom, then the match succeeds only if the input is `eql` to that atom. (4) If the pattern is a conditional like `(?is n numberp)`, then we generate the test `(numberp n)`. Other such patterns could be included here but have not been, since they have not been used. Finally, (5) if the pattern is a list, we check that it has the right operator and arguments.

```
(defun compile-exp (var pattern consequent)
  "Compile code that tests the expression, and does consequent
  if it matches. Assumes bindings in *bindings*."
  (cond ((get-binding pattern *bindings*)
          ;; Test a previously bound variable
          '(if (equal ,var ,(lookup pattern *bindings*))
              ,(force consequent)))
        ((variable-p pattern)
         ;; Add a new bindings; do type checking if needed.
         (push (cons pattern var) *bindings*)
         (force consequent))
        ((atom pattern)
         ;; Match a literal atom
         '(if (eql ,var ',pattern)
              ,(force consequent)))
        ((starts-with pattern '?is)
         (push (cons (second pattern) var) *bindings*)
         '(if (,(third pattern) ,var)
              ,(force consequent)))
        ;; So, far, only the ?is pattern is covered, because
        ;; it is the only one used in simplification rules.
        ;; Other patterns could be compiled by adding code here.
        ;; Or we could switch to a data-driven approach.
        (t ;;; Check the operator and arguments
         '(if (op? ,var ',(exp-op pattern))
              ,(compile-args var pattern consequent))))))
```

The function `compile-args` is used to check the arguments to a pattern. It generates a `let` form binding one or two new variables (for a unary or binary expression), and then calls `compile-exp` to generate code that actually makes the tests. It just passes along the continuation, `consequent`, to `compile-exp`.

```
(defun compile-args (var pattern consequent)
  "Compile code that checks the arg or args, and does consequent
  if the arg(s) match."
  ;;= First make up variable names for the arg(s).
  (let ((,L (symbol var 'L))
        (,R (symbol var 'R)))
    (if (exp-rhs pattern)
        ;;= two arg case
        '(let ((,L (exp-lhs ,var))
              (,R (exp-rhs ,var)))
           ,(compile-exp L (exp-lhs pattern)
                         ,(delay
                            (compile-exp R (exp-rhs pattern)
                                         consequent))))
        ;;= one arg case
        '(let ((,L (exp-lhs ,var)))
           ,(compile-exp L (exp-lhs pattern) consequent)))))
```

The remaining functions are simpler. build-exp generates code to build the right-hand side of a rule, op? tests if its first argument is an expression with a given operator, and symbol constructs a new symbol. Also given is new-symbol, although it is not used in this program.

```
(defun build-exp (exp bindings)
  "Compile code that will build the exp, given the bindings."
  (cond ((assoc exp bindings) (rest (assoc exp bindings)))
        ((variable-p exp)
         (error "Variable ~a occurred on right-hand side,~"
                "but not left." exp))
        ((atom exp) ',exp)
        (t (let ((new-exp (mapcar #'(lambda (x)
                                       (build-exp x bindings))
                                    exp)))
            '(simplify-exp (list ,new-exp))))))

(defun op? (exp op)
  "Does the exp have the given op as its operator?"
  (and (exp-p exp) (eq (exp-op exp) op)))

(defun symbol (&rest args)
  "Concatenate symbols or strings to form an interned symbol"
  (intern (format nil "~{~a~}" args)))

(defun new-symbol (&rest args)
  "Concatenate symbols or strings to form an uninterned symbol"
  (make-symbol (format nil "~{~a~}" args)))
```

Here are some examples of the compiler:

```
> (compile-rule '(= (log (^ e x)) x))
(LAMBDA (X)
  (IF (OP? X 'LOG)
    (LET ((XL (EXP-LHS X)))
      (IF (OP? XL '^)
        (LET ((XLL (EXP-LHS XL))
              (XLR (EXP-RHS XL)))
          (IF (EQL XLL 'E)
            XLR))))))
 
> (compile-rule (simp-rule '(n * (m * x) = (n * m) * x)))
(LAMBDA (X)
  (IF (OP? X '**)
    (LET ((XL (EXP-LHS X))
          (XR (EXP-RHS X)))
      (IF (NUMBERP XL)
        (IF (OP? XR '**)
          (LET ((XRL (EXP-LHS XR))
                (XRR (EXP-RHS XR)))
            (IF (NUMBERP XRL)
              (SIMPLIFY-EXP
                (LIST '**
                  (SIMPLIFY-EXP (LIST '* XL XRL))
                  XRR))))))))
```

### *The Rule-Set Compiler*

The next step is to combine the code generated by this single-rule compiler to generate more compact code for sets of rules. We'll divide up the complete set of rules into subsets based on the main operator (as we did with the `rules-for` function), and generate one big function for each operator. We need to preserve the order of the rules, so only certain optimizations are possible, but if we make the assumption that no function has side effects (a safe assumption in this application), we can still do pretty well. We'll use the `simp-fn` facility to install the one big function for each operator.

The function `compile-rule-set` takes an operator, finds all the rules for that operator, and compiles each rule individually. (It uses `compile-indexed-rule` rather than `compile-rule`, because it assumes we have already done the indexing for the main operator.) After each rule has been compiled, they are combined with `combine-rules`, which merges similar parts of rules and concatenates the different parts. The result is wrapped in a `lambda` expression and compiled as the final simplification function for the operator.

```
(defun compile-rule-set (op)
  "Compile all rules indexed under a given main op,
  and make them into the simp-fn for that op."
  (set-simp-fn op
    (compile nil
      '(lambda (x)
        ,(reduce #'combine-rules
          (mapcar #'compile-indexed-rule
            (rules-for op)))))))

(defun compile-indexed-rule (rule) .
  "Compile one rule into lambda-less code,
  assuming indexing of main op."
  (let ((*bindings* nil))
    (compile-args
      'x (exp-lhs rule)
      (delay (build-exp (exp-rhs rule) *bindings*)))))
```

Here are two examples of what `compile-indexed-rule` generates:

```
> (compile-indexed-rule '(= (log 1) 0))
(LET ((XL (EXP-LHS X)))
  (IF (EQL XL '1)
    '0))

> (compile-indexed-rule '(= (log (^ e x)) x))
(LET ((XL (EXP-LHS X)))
  (IF (OP? XL '^)
    (LET ((XLL (EXP-LHS XL))
          (XLR (EXP-RHS XL)))
      (IF (EQL XLL 'E)
        XLR))))
```

The next step is to combine several of these rules into one. The function `combine-rules` takes two rules and merges them together as much as possible.

```
(defun combine-rules (a b)
  "Combine the code for two rules into one, maintaining order."
  ;; In the default case, we generate the code (or a b),
  ;; but we try to be cleverer and share common code,
  ;; on the assumption that there are no side-effects.
  (cond ((and (listp a) (listp b)
              (= (length a) (length b) 3)
              (equal (first a) (first b))
              (equal (second a) (second b)))
         ;; a=(f x y), b=(f x z) => (f x (combine-rules y z))
         ;; This can apply when f=IF or f=LET
```

```

(list (first a) (second a)
      (combine-rules (third a) (third b))))
((matching-ifs a b)
 '(if ,(second a)
      ,(combine-rules (third a) (third b))
      ,(combine-rules (fourth a) (fourth b))))
 ((starts-with a 'or)
  :: a=(or ... (if p y)), b=(if p z) =>
  :: (or ... (if p (combine-rules y z)))
  :: else
  :: a=(or ...) b => (or ... b)
  (if (matching-ifs (last1 a) b)
      (append (butlast a)
              (list (combine-rules (last1 a) b)))
      (append a (list b))))
  (t ;; a, b => (or a b)
   '(or ,a ,b)))

(defun matching-ifs (a b)
  "Are a and b if statements with the same predicate?"
  (and (starts-with a 'if) (starts-with b 'if)
       (equal (second a) (second b)))))

(defun last1 (list)
  "Return the last element (not last cons cell) of list"
  (first (last list)))

```

Here is what `combine-rules` does with the two rules generated above:

```

> (combine-rules
  '(let ((x1 (exp-lhs x))) (if (eql x1 '1) '0))
  '(let ((x1 (exp-lhs x)))
    (if (op? x1 '^)
        (let ((x1l (exp-lhs x1))
              (x1r (exp-rhs x1)))
          (if (eql x1l 'e) x1r)))))

(LET ((XL (EXP-LHS X)))
  (OR (IF (EQL XL '1) '0)
      (IF (OP? XL '^)
          (LET ((XLL (EXP-LHS XL))
                (XLR (EXP-RHS XL)))
            (IF (EQL XLL 'E) XLR)))))


```

Now we run the compiler by calling `compile-all-rules-indexed` and show the combined compiled simplification function for `log`. The comments were entered by hand to show what simplification rules are compiled where.

```
(defun compile-all-rules-indexed (rules)
  "Compile a separate fn for each operator, and store it
  as the simp-fn of the operator."
  (index-rules rules)
  (let ((all-ops (delete-duplicates (mapcar #'main-op rules))))
    (mapc #'compile-rule-set all-ops))

> (compile-all-rules-indexed *simplification-rules*)
(SIN COS LOG ^ * / - + D)

> (simp-fn 'log)
(LAMBDA (X)
  (LET ((XL (EXP-LHS X)))
    (OR (IF (EQL XL '1)
            '0) ; log 1 = 0
        (IF (EQL XL '0)
            'UNDEFINED) ; log 0 = undefined
        (IF (EQL XL 'E)
            '1) ; log e = 1
        (IF (OP? XL '^)
            (LET ((XLL (EXP-LHS XL))
                  (XLR (EXP-RHS XL)))
              (IF (EQL XLL 'E)
                  XLR)))))) ; log ex = x
```

If we want to bypass the rule-based simplifier altogether, we can change `simplify-exp` once again to eliminate the check for rules:

```
(defun simplify-exp (exp)
  "Simplify by doing arithmetic, or by using the simp function
  supplied for this operator. Do not use rules of any kind."
  (cond ((simplify-by-fn exp))
        ((evaluable exp) (eval exp))
        (t exp)))
```

At last, we are in a position to run the benchmark test on the new compiled code; the function `test-it` runs in about .15 seconds with memoization and .05 without. Why would memoization, which helped before, now hurt us? Probably because there is a lot of overhead in accessing the hash table, and that overhead is only worth it when there is a lot of other computation to do.

We've seen a great improvement since the original code, as the following table summarizes. Overall, the various efficiency improvements have resulted in a 130-fold speed-up—we can do now in a minute what used to take two hours. Of course, one must keep in mind that the statistics are only good for this one particular set of

test data on this one machine. It is an open question what performance you will get on other problems and on other machines.

The following table summarizes the execution time and number of function calls on the test data:

	original	memo	memo+index	memo+comp	comp
run time (secs)	6.6	3.0	.98	.15	.05
speed-up	—	2	7	44	130
<hr/>					
calls					
pat-match	51690	20003	5159	0	0
variable-p	37908	14694	4798	0	0
match-variable	1393	551	551	0	0
simplify	906	408	408	545	906
simplify-exp	274	118	118	118	274

## 9.7 History and References

The idea of memoization was introduced by Donald Michie 1968. He proposed using a list of values rather than a hash table, so the savings was not as great. In mathematics, the field of dynamic programming is really just the study of how to compute values in the proper order so that partial results will already be cached away when needed.

A large part of academic computer science covers compilation; Aho and Ullman 1972 is just one example. The technique of compiling embedded languages (such as the language of pattern-matching rules) is one that has achieved much more attention in the Lisp community than in the rest of computer science. See Emanuelson and Haraldsson 1980, for an example.

Choosing the right data structure, indexing it properly, and defining algorithms to operate on it is another important branch of computer science; Sedgewick 1988 is one example, but there are many worthy texts.

Delaying computation by packaging it up in a `lambda` expression is an idea that goes back to Algol's use of *thunks*—a mechanism to implement call-by-name parameters, essentially by passing functions of no arguments. The name *thunk* comes from the fact that these functions can be compiled: the system does not have to think about them at run time, because the compiler has already thunked about them. Peter Ingberman 1961 describes thunks in detail. Abelson and Sussman 1985 cover delays nicely. The idea of eliminating unneeded computation is so attractive that entire languages have built around the concept of *lazy evaluation*—don't evaluate an expression until its value is needed. See Hughes 1985 or Field and Harrison 1988.

## 9.8 Exercises

- ?
- Exercise 9.3 [d]** In this chapter we presented a compiler for `simplify`. It is not too much harder to extend this compiler to handle the full power of `pat-match`. Instead of looking at expressions only, allow trees with variables in any position. Extend and generalize the definitions of `compile-rule` and `compile-rule-set` so that they can be used as a general tool for any application program that uses `pat-match` and/or `rule-based-translator`. Make sure that the compiler is data-driven, so that the programmer who adds a new kind of pattern to `pat-match` can also instruct the compiler how to deal with it. One hard part will be accounting for segment variables. It is worth spending a considerable amount of effort at compile time to make this efficient at run time.
  
- ?
- Exercise 9.4 [m]** Define the time to compute  $(\text{fib } n)$  without memoization as  $T_n$ . Write a formula to express  $T_n$ . Given that  $T_{25} \approx 1.1$  seconds, predict  $T_{100}$ .
  
- ?
- Exercise 9.5 [m]** Consider a version of the game of Nim played as follows: there is a pile of  $n$  tokens. Two players alternate removing tokens from the pile; on each turn a player must take either one, two, or three tokens. Whoever takes the last token wins. Write a program that, given  $n$ , returns the number of tokens to take to insure a win, if possible. Analyze the execution times for your program, with and without memoization.
  
- ?
- Exercise 9.6 [m]** A more complicated Nim-like game is known as Grundy's game. The game starts with a single pile of  $n$  tokens. Each player must choose one pile and split it into two uneven piles. The first player to be unable to move loses. Write a program to play Grundy's game, and see how memoization helps.
  
- ?
- Exercise 9.7 [h]** This exercise describes a more challenging one-person game. In this game the player rolls a six-sided die eight times. The player forms four two-digit decimal numbers such that the total of the four numbers is as high as possible, but not higher than 170. A total of 171 or more gets scored as zero.

The game would be deterministic and completely boring if not for the requirement that after each roll the player must immediately place the digit in either the ones or tens column of one of the four numbers.

Here is a sample game. The player first rolls a 3 and places it in the ones column of the first number, then rolls a 4 and places it in the tens column, and so on. On the last roll the player rolls a 6 and ends up with a total of 180. Since this is over the limit of 170, the player's final score is 0.

roll	3	4	6	6	3	5	3	6
1st num.	-3	43	43	43	43	43	43	43
2nd num.	-	-	-6	-6	36	36	36	36
3rd num.	-	-	-	-6	-6	-6	36	36
4th num.	-	-	-	-	-	-5	-5	65
total	03	43	49	55	85	90	120	0

Write a function that allows you to play a game or a series of games. The function should take as argument a function representing a strategy for playing the game.

- ❑ **Exercise 9.8 [h]** Define a good strategy for the dice game described above. (Hint: my strategy scores an average of 143.7.)
- ❑ **Exercise 9.9 [m]** One problem with playing games involving random numbers is the possibility that a player can cheat by figuring out what `random` is going to do next. Read the definition of the function `random` and describe how a player could cheat. Then describe a countermeasure.
- ❑ **Exercise 9.10 [m]** On page 292 we saw the use of the read-time conditionals, `#+` and `#-`, where `#+` is the read-time equivalent of `when`, and `#-` is the read-time equivalent of `unless`. Unfortunately, there is no read-time equivalent of `case`. Implement one.
- ❑ **Exercise 9.11 [h]** Write a compiler for ELIZA that compiles all the rules at once into a single function. How much more efficient is the compiled version?
- ❑ **Exercise 9.12 [d]** Write some rules to simplify Lisp code. Some of the algebraic simplification rules will still be valid, but new ones will be needed to simplify nonalgebraic functions and special forms. (Since `nil` is a valid expression in this domain, you will have to deal with the semipredicate problem.) Here are some example rules (using prefix notation):

```
(= (+ x 0) x)
(= 'nil nil)
(= (car (cons x y)) x)
(= (cdr (cons x y)) y)
(= (if t x y) x)
(= (if nil x y) y)
(= (length nil) 0)
(= (expt y (?if x numberp)) (expt (expt y (/ x 2)) 2))
```

 **Exercise 9.13 [m]** Consider the following two versions of the sieve of Eratosthenes algorithm. The second explicitly binds a local variable. Is this worth it?

```
(defun sieve (pipe)
  (make-pipe (head pipe)
    (filter #'(lambda (x) (/= (mod x (head pipe)) 0))
      (sieve (tail pipe)))))

(defun sieve (pipe)
  (let ((first-num (head pipe)))
    (make-pipe first-num
      (filter #'(lambda (x) (/= (mod x first-num) 0))
        (sieve (tail pipe))))))
```

## 9.9 Answers

**Answer 9.4** Let  $F_n$  denote (`fib n`). Then the time to compute  $F_n$ ,  $T_n$ , is a small constant for  $n \leq 1$ , and is roughly equal to  $T_{n-1}$  plus  $T_{n-2}$  for larger  $n$ . Thus,  $T_n$  is roughly proportional to  $F_n$ :

$$T_n = F_n \frac{T_i}{F_i}$$

We could use some small value of  $T_i$  to calculate  $T_{100}$  if we knew  $F_{100}$ . Fortunately, we can use the equation:

$$F_n \propto \phi^n$$

where  $\phi = (1 + \sqrt{5})/2 \approx 1.618$ . This equation was derived by de Moivre in 1718 (see Knuth, Donald E. *Fundamental Algorithms*, pp. 78–83), but the number  $\phi$  has a long interesting history. Euclid called it the “extreme and mean ratio,” because the ratio of  $A$  to  $B$  is the ratio of  $A + B$  to  $A$  if  $A/B$  is  $\phi$ . In the Renaissance it was called the “divine proportion,” and in the last century it has been known as the “golden ratio,” because a rectangle with sides in this ratio can be divided into two smaller rectangles that both have the same ratio between sides. It is said to be a pleasing proportion when employed in paintings and architecture. Putting history aside, given  $T_{25} \approx 1.1\text{sec}$  we can now calculate:

$$T_{100} \approx \phi^{100} \frac{1.1\text{sec}}{\phi^{25}} \approx 5 \times 10^{15} \text{sec}$$

which is roughly 150 million years. We can also see that the timing data in the table fits the equation fairly well. However, we would expect some additional time for larger numbers because it takes longer to add and garbage collect bignums than fixnums.

**Answer 9.5** First we'll define the notion of a forced win. This occurs either when there are three or fewer tokens left or when you can make a move that gives your opponent a possible loss. A possible loss is any position that is not a forced win. If you play perfectly, then a possible loss for your opponent will in fact be a win for you, since there are no ties. See the functions `win` and `loss` below. Now your strategy should be to win the game outright if there are three or fewer tokens, or otherwise to choose the largest number resulting in a possible loss for your opponent. If there is no such move available to you, take only one, on the grounds that your opponent is more likely to make a mistake with a larger pile to contend with. This strategy is embodied in the function `nim` below.

```
(defun win (n)
  "Is a pile of n tokens a win for the player to move?"
  (or (<= n 3)
      (loss (- n 1))
      (loss (- n 2))
      (loss (- n 3)))))

(defun loss (n) (not (win n)))

(defun nim (n)
  "Play Nim: a player must take 1-3; taking the last one wins."
  (cond ((<= n 3) n) ; an immediate win
        ((loss (- n 3)) 3) ; an eventual win
        ((loss (- n 2)) 2) ; an eventual win
        ((loss (- n 1)) 1) ; an eventual win
        (t 1))) ; a loss; the 1 is arbitrary

(memoize 'loss)
```

From this we are able to produce a table of execution times (in seconds), with and without memoization. Only `loss` need be memoized. (Why?) Do you have a good explanation of the times for the unmemoized version? What happens if you change the order of the loss clauses in `win` and/or `nim`?

**Answer 9.6** We start by defining a function, `moves`, which generates all possible moves from a given position. This is done by considering each pile of  $n$  tokens within a set of piles  $s$ . Any pile bigger than two tokens can be split. We take care to eliminate duplicate positions by sorting each set of piles, and then removing the duplicates.

```
(defun moves (s)
  "Return a list of all possible moves in Grundy's game"
  ;;= S is a list of integers giving the sizes of the piles
  (remove-duplicates
    (loop for n in s append (make-moves n s))
    :test #'equal))
```

```
(defun make-moves (n s)
  (when (>= n 2)
    (let ((s/n (remove n s :count 1)))
      (loop for i from 1 to (- (ceiling n 2) 1)
            collect (sort* (list* i (- n i) s/n)
                           #'>)))))

(defun sort* (seq pred &key key)
  "Sort without altering the sequence"
  (sort (copy-seq seq) pred :key key))
```

This time a loss is defined as a position from which you have no moves, or one from which your opponent can force a win no matter what you do. A winning position is one that is not a loss, and the strategy is to pick a move that is a loss for your opponent, or if you can't, just to play anything (here we arbitrarily pick the first move generated).

```
(defun loss (s)
  (let ((choices (moves s)))
    (or (null choices)
        (every #'win choices)))

(defun win (s) (not (loss s)))

(defun grundy (s)
  (let ((choices (moves s)))
    (or (find-if #'loss choices)
        (first choices))))
```

**Answer 9.7** The answer assumes that a strategy function takes four arguments: the current die roll, the score so far, the number of remaining positions in the tens column, and the number of remaining positions in the ones column. The strategy function should return 1 or 10.

```
(defun play-games (&optional (n-games 10) (player 'make-move))
  "A driver for a simple dice game. In this game the player
  rolls a six-sided die eight times. The player forms four
  two-digit decimal numbers such that the total of the four
  numbers is as high as possible, but not higher than 170.
  A total of 171 or more gets scored as zero. After each die
  is rolled, the player must decide where to put it.
  This function returns the player's average score."
  (/ (loop repeat n-games summing (play-game player 0 4 4))
      (float n-games)))
```

---

```
(defun play-game (player &optional (total 0) (tens 4) (ones 4))
  (cond ((or (> total 170) (< tens 0) (< ones 0)) 0)
        ((and (= tens 0) (= ones 0)) total)
        (t (let ((die (roll-die)))
             (case (funcall player die total tens ones)
               (1 (play-game player (+ total die)
                             tens (- ones 1)))
               (10 (play-game player (+ total (* 10 die))
                            (- tens 1) ones))
               (t 0))))))

(defun roll-die () (+ 1 (random 6)))
```

So, the expression `(play-games 5 #'make-move)` would play five games with a strategy called `make-move`. This returns only the average score of the games; if you want to see each move as it is played, use this function:

```
(defun show (player)
  "Return a player that prints out each move it makes."
  #'(lambda (die total tens ones)
      (when (= total 0) (fresh-line))
      (let ((move (funcall player die total tens ones)))
        (incf total (* die move))
        (format t "~2d->~3d | ~@[*~]" (* move die) total (> total 170))
        move)))
```

and call `(play-games 5 (show #'make-moves))`.

**Answer 9.9** The expression `(random 6 (make-random-state))` returns the next number that `roll-die` will return. To guard against this, we can make `roll-die` use a random state that is not accessible through a global variable:

```
(let ((state (make-random-state t)))
  (defun roll-die () (+ 1 (random 6 state))))
```

**Answer 9.10** Because this has to do with read-time evaluation, it must be implemented as a macro or read macro. Here's one way to do it:

```
(defmacro read-time-case (first-case &rest other-cases)
  "Do the first case, where normally cases are
  specified with #+ or possibly #- marks."
  (declare (ignore other-cases))
  first-case)
```

A fanciful example, resurrecting a number of obsolete Lisps, follows:

```
(defun get-fast-time ()
  (read-time-case
    #+Explorer      (time:microsecond-time)
    #+Franz        (sys:time)
    #+(or PSL UCI) (time)
    #+YKT          (currenttime)
    #+MTS          (status 39)
    #+Interlisp    (clock 1)
    #+Lisp1.5      (tempus-fugit)
    ;;= otherwise
                (get-internal-real-time)))
```

**Answer 9.13** Yes. Computing (`head pipe`) may be a trivial computation, but it will be done many times. Binding the local variable makes sure that it is only done once. In general, things that you expect to be done multiple times should be moved out of delayed functions, while things that may not be done at all should be moved inside a delay.

## CHAPTER **10**

---

# Low-Level Efficiency Issues

*There are only two qualities in the world: efficiency and inefficiency; and only two sorts of people: the efficient and the inefficient.*

—George Bernard Shaw  
*John Bull's Other Island* (1904)

The efficiency techniques of the previous chapter all involved fairly significant changes to an algorithm. But what happens when you already are using the best imaginable algorithms, and performance is still a problem? One answer is to find what parts of the program are used most frequently and make micro-optimizations to those parts. This chapter covers the following six optimization techniques. If your programs all run quickly enough, then feel free to skip this chapter. But if you would like your programs to run faster, the techniques described here can lead to speed-ups of 40 times or more.

- Use declarations.
- Avoid generic functions.
- Avoid complex argument lists.
- Provide compiler macros.
- Avoid unnecessary consing.
- Use the right data structure.

## 10.1 Use Declarations

On general-purpose computers running Lisp, much time is spent on type-checking. You can gain efficiency at the cost of robustness by declaring, or promising, that certain variables will always be of a given type. For example, consider the following function to compute the sum of the squares of a sequence of numbers:

```
(defun sum-squares (seq)
  (let ((sum 0))
    (dotimes (i (length seq))
      (incf sum (square (elt seq i))))
    sum))

(defun square (x) (* x x))
```

If this function will only be used to sum vectors of fixnums, we can make it a lot faster by adding declarations:

```
(defun sum-squares (vect)
  (declare (type (simple-array fixnum *) vect)
           (inline square) (optimize speed (safety 0)))
  (let ((sum 0))
    (declare (fixnum sum))
    (dotimes (i (length vect))
      (declare (fixnum i))
      (incf sum (the fixnum (square (svref vect i)))))))
  sum))
```

The fixnum declarations let the compiler use integer arithmetic directly, rather than checking the type of each addend. The `(the fixnum ...)` special form is a promise that the argument is a fixnum. The `(optimize speed (safety 0))` declaration tells the compiler to make the function run as fast as possible, at the possible expense of

making the code less safe (by ignoring type checks and so on). Other quantities that can be optimized are compilation-speed, space and in ANSI Common Lisp only, debug (ease of debugging). Quantities can be given a number from 0 to 3 indicating how important they are; 3 is most important and is the default if the number is left out.

The (`inline square`) declaration allows the compiler to generate the multiplication specified by `square` right in the loop, without explicitly making a function call to `square`. The compiler will create a local variable for `(svref vect i)` and will not execute the reference twice—inline functions do not have any of the problems associated with macros as discussed on page 853. However, there is one drawback: when you redefine an inline function, you may need to recompile all the functions that call it.

You should declare a function `inline` when it is short and the function-calling overhead will thus be a significant part of the total execution time. You should not declare a function `inline` when the function is recursive, when its definition is likely to change, or when the function's definition is long and it is called from many places.

In the example at hand, declaring the function `inline` saves the overhead of a function call. In some cases, further optimizations are possible. Consider the predicate `starts-with`:

```
(defun starts-with (list x)
  "Is this a list whose first element is x?"
  (and (consp list) (eql (first list) x)))
```

Suppose we have a code fragment like the following:

```
(if (consp list) (starts-with list x) ...)
```

If `starts-with` is declared `inline` this will expand to:

```
(if (consp list) (and (consp list) (eql (first list) x)) ...)
```

which many compilers will simplify to:

```
(if (consp list) (eql (first list) x) ...)
```

Very few compilers do this kind of simplification across functions without the hint provided by `inline`.

Besides eliminating run-time type checks, declarations also allow the compiler to choose the most efficient representation of data objects. Many compilers support both *boxed* and *unboxed* representations of data objects. A boxed representation includes enough information to determine the type of the object. An unboxed representation is just the “raw bits” that the computer can deal with directly. Consider

the following function, which is used to clear a  $1024 \times 1024$  array of floating point numbers, setting each one to zero:

```
(defun clear-m-array (array)
  (declare (optimize (speed 3) (safety 0)))
  (declare (type (simple-array single-float (1024 1024)) array))
  (dotimes (i 1024)
    (dotimes (j 1024)
      (setf (aref array i j) 0.0))))
```

In Allegro Common Lisp on a Sun SPARCstation, this compiles into quite good code, comparable to that produced by the C compiler for an equivalent C program. If the declarations are omitted, however, the performance is about 40 times worse.

The problem is that without the declarations, it is not safe to store the raw floating point representation of 0.0 in each location of the array. Instead, the program has to box the 0.0, allocating storage for a typed pointer to the raw bits. This is done inside the nested loops, so the result is that each call to the version of `clear-m-array` without declarations calls the floating-point-boxing function 1048567 times, allocating a megaword of storage. Needless to say, this is to be avoided.

Not all compilers heed all declarations; you should check before wasting time with declarations your compiler may ignore. The function `disassemble` can be used to show what a function compiles into. For example, consider the trivial function to add two numbers together. Here it is with and without declarations:

```
(defun f (x y)
  (declare (fixnum x y) (optimize (safety 0) (speed 3)))
  (the fixnum (+ x y)))

(defun g (x y) (+ x y))
```

Here is the disassembled code for `f` from Allegro Common Lisp for a Motorola 68000-series processor:

```
> (disassemble 'f)
;; disassembling #<Function f @ #x83ef79>
;; formals: x y
;; code vector @ #x83ef44
0:     link   a6,#0
4:     move.l a2,-(a7)
6:     move.l a5,-(a7)
8:     move.l 7(a2),a5
12:    move.l 8(a6),d4 ; y
16:    add.l  12(a6),d4 ; x
20:    move.l #1,d1
```

```
22:    move.l  -8(a6),a5
26:    unlk   a6
28:    rtd    #8
```

This may look intimidating at first glance, but you don't have to be an expert at 68000 assembler to gain some appreciation of what is going on here. The instructions labeled 0-8 (labels are in the leftmost column) comprise the typical function preamble for the 68000. They do subroutine linkage and store the new function object and constant vector into registers. Since f uses no constants, instructions 6, 8, and 22 are really unnecessary and could be omitted. Instructions 0, 4, and 26 could also be omitted if you don't care about seeing this function in a stack trace during debugging. More recent versions of the compiler will omit these instructions.

The heart of function f is the two-instruction sequence 12-16. Instruction 12 retrieves y, and 16 adds y to x, leaving the result in d4, which is the "result" register. Instruction 20 sets d1, the "number of values returned" register, to 1.

Contrast this to the code for g, which has no declarations and is compiled at default speed and safety settings:

```
> (disassemble 'g)
;; disassembling #<Function g @ #x83dbd1>
;; formals: x y
;; code vector @ #x83db64
0:    add.l  #8,31(a2)
4:    sub.w  #2,d1
6:    beq.s  12
8:    jmp    16(a4)  ; wnaerr
12:   link   a6,#0
16:   move.l a2,-(a7)
18:   move.l a5,-(a7)
20:   move.l 7(a2),a5
24:   tst.b  -208(a4) ; signal-hit
28:   beq.s  34
30:   jsr    872(a4)  ; process-sig
34:   move.l 8(a6),d4  ; y
38:   move.l 12(a6),d0  ; x
42:   or.l   d4,d0
44:   and.b  #7,d0
48:   bne.s  62
50:   add.l  12(a6),d4  ; x
54:   bvc.s  76
56:   jsr    696(a4)  ; add-overflow
60:   bra.s  76
62:   move.l 12(a6),-(a7) ; x
66:   move.l d4,-(a7)
68:   move.l #2,d1
```

```

70:      move.l  -304(a4),a0 ; +_2op
74:      jsr     (a4)
76:      move.l  #1,d1
78:      move.l  -8(a6),a5
82:      unlk   a6
84:      rtd    #8

```

See how much more work is done. The first four instructions ensure that the right number of arguments have been passed to g. If not, there is a jump to wnaerr (wrong-number-of-arguments-error). Instructions 12–20 have the argument loading code that was at 0–8 in f. At 24–30 there is a check for asynchronous signals, such as the user hitting the abort key. After x and y are loaded, there is a type check (42–48). If the arguments are not both fixnums, then the code at instructions 62–74 sets up a call to +\_2op, which handles type coercion and non-fixnum addition. If all goes well, we don't have to call this routine, and do the addition at instruction 50 instead. But even then we are not done—just because the two arguments were fixnums does not mean the result will be. Instructions 54–56 check and branch to an overflow routine if needed. Finally, instructions 76–84 return the final value, just as in f.

Some low-quality compilers ignore declarations altogether. Other compilers don't need certain declarations, because they can rely on special instructions in the underlying architecture. On a Lisp Machine, both f and g compile into the same code:

6 PUSH	ARG 0 : X
7 +	ARG 1 : Y
8 RETURN	PDL-POP

The Lisp Machine has a microcoded + instruction that simultaneously does a fixnum add and checks for non-fixnum arguments, branching to a subroutine if either argument is not a fixnum. The hardware does the work that the compiler has to do on a conventional processor. This makes the Lisp Machine compiler simpler, so compiling a function is faster. However, on modern pipelined computers with instruction caches, there is little or no advantage to microcoding. The current trend is away from microcode toward reduced instruction set computers (RISC).

On most computers, the following declarations are most likely to be helpful:

- **fixnum** and **float**. Numbers declared as fixnums or floating-point numbers can be handled directly by the host computer's arithmetic instructions. On some systems, float by itself is not enough; you have to say `single-float` or `double-float`. Other numeric declarations will probably be ignored. For example, declaring a variable as `integer` does not help the compiler much, because bignums are integers. The code to add bignums is too complex to put

inline, so the compiler will branch to a general-purpose routine (like `+_2op` in Allegro), the same routine it would use if no declarations were given.

- `list` and `array`. Many Lisp systems provide separate functions for the list- and array- versions of commonly used sequence functions. For example, `(delete x (the list l))` compiles into `(sys:delete-list-eql x l)` on a TI Explorer Lisp Machine. Another function, `sys:delete-vector`, is used for arrays, and the generic function `delete` is used only when the compiler can't tell what type the sequence is. So if you know that the argument to a generic function is either a list or an array, then declare it as such.
- `simple-vector` and `simple-array`. Simple vectors and arrays are those that do not share structure with other arrays, do not have fill pointers, and are not adjustable. In many implementations it is faster to `aref` a `simple-vector` than a `vector`. It is certainly much faster than taking an `elt` of a sequence of unknown type. Declare your arrays to be simple (if they in fact are).
- `(array type)`. It is often important to specialize the type of array elements. For example, an `(array short-float)` may take only half the storage of a general array, and such a declaration will usually allow computations to be done using the CPU's native floating-point instructions, rather than converting into and out of Common Lisp's representation of floating points. This is very important because the conversion normally requires allocating storage, but the direct computation does not. The specifiers `(simple-array type)` and `(vector type)` should be used instead of `(array type)` when appropriate. A very common mistake is to declare `(simple-vector type)`. This is an error because Common Lisp expects `(simple-vector size)`—don't ask me why.
- `(array * dimensions)`. The full form of an array or `simple-array` type specifier is `(array type dimensions)`. So, for example, `(array bit (* *))` is a two-dimensional bit array, and `(array bit (1024 1024))` is a  $1024 \times 1024$  bit array. It is very important to specify the number of dimensions when known, and less important to specify the exact size, although with multidimensional arrays, declaring the size is more important. The format for a `vector` type specifier is `(vector type size)`.

Note that several of these declarations can apply all at once. For example, in

```
(position #\.\. (the simple-string file-name))
```

the variable `filename` has been declared to be a vector, a simple array, and a sequence of type `string-char`. All three of these declarations are helpful. The type `simple-string` is an abbreviation for `(simple-array string-char)`.

This guide applies to most Common Lisp systems, but you should look in the implementation notes for your particular system for more advice on how to fine-tune your code.

## 10.2 Avoid Generic Functions

Common Lisp provides functions with great generality, but someone must pay the price for this generality. For example, if you write `(elt x 0)`, different machine instruction will be executed depending on if `x` is a list, string, or vector. Without declarations, checks will have to be done at runtime. You can either provide declarations, as in `(elt (the list x) 0)`, or use a more specific function, such as `(first x)` in the case of lists, `(char x 0)` for strings, `(aref x 0)` for vectors, and `(svref x 0)` for simple vectors. Of course, generic functions are useful—I wrote `random-elt` as shown following to work on lists, when I could have written the more efficient `random-mem` instead. The choice paid off when I wanted a function to choose a random character from a string—`random-elt` does the job unchanged, while `random-mem` does not.

```
(defun random-elt (s) (elt s (random (length s))))
(defun random-mem (l) (nth (random (length (the list l))) l))
```

This example was simple, but in more complicated cases you can make your sequence functions more efficient by having them explicitly check if their arguments are lists or vectors. See the definition of `map-into` on page 857.

## 10.3 Avoid Complex Argument Lists

Functions with keyword arguments suffer a large degree of overhead. This may also be true for optional and rest arguments, although usually to a lesser degree. Let's look at some simple examples:

```
(defun reg (a b c d) (list a b c d))
(defun rst (a b c &rest d) (list* a b c d))
(defun opt (&optional a b (c 1) (d (sqrt a))) (list a b c d))
(defun key (&key a b (c 1) (d (sqrt a))) (list a b c d))
```

We can see what these compile into for the TI Explorer, but remember that your compiler may be quite different.

```

> (disassemble 'reg)
  8 PUSH      ARG!0    ; A
  9 PUSH      ARG!1    ; B
 10 PUSH     ARG!2    ; C
 11 PUSH     ARG!3    ; D
 12 TAIL-REC CALL-4 FEF!3  ; #'LIST

> (disassemble 'rst)
  8 PUSH      ARG!0    ; A
  9 PUSH      ARG!1    ; B
 10 PUSH     ARG!2    ; C
 11 PUSH     LOCAL!0  ; D
 12 RETURN CALL-4 FEF!3  ; #'LIST*

```

With the regular argument list, we just push the four variables on the argument stack and branch to the list function. (Chapter 22 explains why a tail-recursive call is just a branch statement.)

With a rest argument, things are almost as easy. It turns out that on this machine, the microcode for the calling sequence automatically handles rest arguments, storing them in local variable 0. Let's compare with optional arguments:

```

(defun opt (&optional a b (c 1) (d (sqrt a))) (list a b c d))

> (disassemble 'opt)
 24 DISPATCH   FEF!5    ; [0⇒25;1⇒25;2⇒25;3⇒27;ELSE⇒30]
 25 PUSH-NUMBER 1
 26 POP        ARG!2    ; C
 27 PUSH      ARG!0    ; A
 28 PUSH CALL-1 FEF!3  ; #'SQRT
 29 POP        ARG!3    ; D
 30 PUSH      ARG!0    ; A
 31 PUSH      ARG!1    ; B
 32 PUSH      ARG!2    ; C
 33 PUSH      ARG!3    ; D
 34 TAI-REC CALL-4 FEF!4  ; #'LIST

```

Although this assembly language may be harder to read, it turns out that optional arguments are handled very efficiently. The calling sequence stores the number of optional arguments on top of the stack, and the DISPATCH instruction uses this to index into a table stored at location FEF!5 (an offset five words from the start of the function). The result is that in one instruction the function branches to just the right place to initialize any unspecified arguments. Thus, a function with optional arguments that are all supplied takes only one more instruction (the dispatch) than the "regular" case. Unfortunately, keyword arguments don't fare as well:

```
(defun key (&key a b (c 1) (d (sqrt a))) (list a b c d))
```

```

> (disassemble 'key)
14 PUSH-NUMBER    1
15 POP            LOCAL|3 ; C
16 PUSH            FEF|3 ; SYS::KEYWORD-GARBAGE
17 POP            LOCAL|4
18 TEST            LOCAL|0
19 BR-NULL     24
20 PUSH            FEF|4 ; '(:A :B :C :D)
21 SET-NIL        PDL-PUSH
22 PUSH-LOC       LOCAL|1 ; A
23 (AUX) %STORE-KEY-WORD-ARGS
24 PUSH            LOCAL|1 ; A
25 PUSH            LOCAL|2 ; B
26 PUSH            LOCAL|3 ; C
27 PUSH            LOCAL|4
28 EQ              FEF|3 ; SYS::KEYWORD-GARBAGE
29 BR-NULL     33
30 PUSH            LOCAL|1 ; A
31 PUSH CALL-1    FEF|5 ; #'SQRT
32 RETURN CALL-4  FEF|6 ; #'LIST
33 PUSH            LOCAL|4
34 RETURN CALL-4  FEF|6 ; #'LIST

```

It is not important to be able to read all this assembly language. The point is that there is considerable overhead, even though this architecture has a specific instruction (%STORE-KEY-WORD-ARGS) to help deal with keyword arguments.

Now let's look at the results on another system, the Allegro compiler for the 68000. First, here's the assembly code for `reg`, to give you an idea of the minimal calling sequence:<sup>1</sup>

```

> (disassemble 'reg)
;; disassembling #<Function reg @ #x83db59>
;; formals: a b c d
;; code vector @ #x83db1c
0:      link   a6,#0
4:      move.l  a2,-(a7)
6:      move.l  a5,-(a7)
8:      move.l  7(a2),a5
12:     move.l  20(a6),-(a7) ; a
16:     move.l  16(a6),-(a7) ; b
20:     move.l  12(a6),-(a7) ; c
24:     move.l  8(a6),-(a7) ; d
28:     move.l  #4,d1
30:     jsr    848(a4)      ; list

```

---

<sup>1</sup>These are all done with safety 0 and speed 3.

```

34:    move.l  -8(a6),a5
38:    unlk   a6
40:    rtd    #10

```

Now we see that `&rest` arguments take a lot more code in this system:

```

> (disassemble 'rst)
;; disassembling #<Function rst @ #x83de89>
;; formals: a b c &rest d
;; code vector @ #x83de34
0:    sub.w  #3,d1
2:    bge.s  8
4:    jmp    16(a4)          ; wnaerr
8:    move.l (a7)+,a1
10:   move.l d3,-(a7)        ; nil
12:   sub.w  #1,d1
14:   blt.s  38
16:   move.l a1,-52(a4)      ; c_protected-retaddr
20:   jsr    40(a4)          ; cons
24:   move.l d4,-(a7)
26:   dbra   d1,20
30:   move.l -52(a4),a1      ; c_protected-retaddr
34:   clr.l  -52(a4)          ; c_protected-retaddr
38:   move.l a1,-(a7)
40:   link   a6,#0
44:   move.l a2,-(a7)
46:   move.l a5,-(a7)
48:   move.l 7(a2),a5
52:   move.l -332(a4),a0      ; list*
56:   move.l -8(a6),a5
60:   unlk   a6
62:   move.l #4,d1
64:   jmp    (a4)

```

The loop from 20-26 builds up the `&rest` list one `cons` at a time. Part of the difficulty is that `cons` could initiate a garbage collection at any time, so the list has to be built in a place that the garbage collector will know about. The function with optional arguments is even worse, taking 34 instructions (104 bytes), and keywords are worst of all, weighing in at 71 instructions (178 bytes), and including a loop. The overhead for optional arguments is proportional to the number of optional arguments, while for keywords it is proportional to the product of the number of parameters allowed and the number of arguments actually supplied.

A good guideline to follow is to use keyword arguments primarily as an interface to infrequently used functions, and to provide versions of these functions without keywords that can be used in places where efficiency is important. Consider:

```
(proclaim '(inline key))
(defun key (&key a b (c 1) (d (sqrt a))) (*no-key a b c d))
(defun *no-key (a b c d) (list a b c d))
```

Here the function `key` is used as an interface to the function `no-key`, which does the real work. The inline proclamation should allow the compiler to compile a call to `key` as a call to `no-key` with the appropriate arguments:

```
> (disassemble #'(lambda (x y) (key :b x :a y)))
10 PUSH      ARG|1    ; Y
11 PUSH      ARG|0    ; X
12 PUSH-NUMBER 1
13 PUSH      ARG|1    ; Y
14 PUSH CALL-1  FEF|3    ; #'SQRT
15 TAIL-REC CALL-4  FEF|4    ; #'NO-KEY
```

The overhead only comes into play when the keywords are not known at compile time. In the following example, the compiler is forced to call `key`, not `no-key`, because it doesn't know what the keyword `k` will be at run time:

```
> (disassemble #'(lambda (k x y) (key k x :a y)))
10 PUSH      ARG|0    ; K
11 PUSH      ARG|1    ; X
12 PUSH      FEF|3    ; ':A
13 PUSH      ARG|2    ; Y
14 TAIL-REC CALL-4  FEF|4    ; #'KEY
```

Of course, in this simple example I could have replaced `no-key` with `list`, but in general there will be some more complex processing. If I had proclaimed `no-key` inline as well, then I would get the following:

```
> (disassemble #'(lambda (x y) (key :b x :a y)))
10 PUSH      ARG|1    ; Y
11 PUSH      ARG|0    ; X
12 PUSH-NUMBER 1
13 PUSH      ARG|1    ; Y
14 PUSH CALL-1  FEF|3    ; #'SQRT
15 TAIL-REC CALL-4  FEF|4    ; #'LIST
```

If you like, you can define a macro to automatically define the interface to the keyword-less function:

```
(defmacro defun* (fn-name arg-list &rest body)
  "Define two functions, one an interface to a &keyword-less
  version. Proclaim the interface function inline."
  (if (and (member '&key arg-list)
            (not (member '&rest arg-list)))
      (let ((no-key-fn-name (symbol fn-name '*no-key))
            (args (mapcar #'first-or-self
                          (set-difference
                           arg-list
                           lambda-list-keywords))))
        '(progn
          (proclaim '(inline ,fn-name))
          (defun ,no-key-fn-name ,args
            ,body)
          (defun ,fn-name ,arg-list
            (,no-key-fn-name ,args)))
        '(defun ,fn-name ,arg-list
            ,body)))
    > (macroexpand '(defun* key (&key a b (c 1) (d (sqrt a)))
                     (list a b c d)))
    (PROGN (PROCLAIM '(INLINE KEY))
           (DEFUN KEY*NO-KEY (A B C D) (LIST A B C D))
           (DEFUN KEY (&KEY A B (C 1) (D (SQRT A)))
             (KEY*NO-KEY A B C D)))
    > (macroexpand '(defun* reg (a b c d) (list a b c d)))
    (DEFUN REG (A B C D) (LIST A B C D))
```

There is one disadvantage to this approach: a user who wants to declare key inline or not inline does not get the expected result. The user has to know that key is implemented with key\*no-key, and declare key\*no-key inline.

An alternative is just to proclaim the function that uses &key to be inline. Rob MacLachlan provides an example. In CMU Lisp, the function member has the following definition, which is proclaimed inline:

```
(defun member (item list &key (key #'identity)
              (test #'eql testp) (test-not nil notp))
  (do ((list list (cdr list)))
      ((null list) nil)
      (let ((car (car list)))
        (if (cond
              (testp
               (funcall test item
                         (funcall key car)))
              (notp
               (not
```

```
(funcall test-not item
          (funcall key car))))
(t
  (funcall test item
            (funcall key car)))
  (return list)))))
```

A call like `(member ch l :key #'first-letter :test #'char=)` expands into the equivalent of the following code. Unfortunately, not all compilers are this clever with inline declarations.

```
(do ((list list (cdr list)))
    ((null list) nil)
    (let ((car (car list)))
      (if (char= ch (first-letter car))
          (return list))))
```

This chapter is concerned with efficiency and so has taken a stand against the use of keyword parameters in frequently used functions. But when maintainability is considered, keyword parameters look much better. When a program is being developed, and it is not clear if a function will eventually need additional arguments, keyword parameters may be the best choice.

## 10.4 Avoid Unnecessary Consing

The `cons` function may appear to execute quite quickly, but like all functions that allocate new storage, it has a hidden cost. When large amounts of storage are used, eventually the system must spend time garbage collecting. We have not mentioned it earlier, but there are actually two relevant measures of the amount of space consumed by a program: the amount of storage allocated, and the amount of storage retained. The difference is storage that is used temporarily but eventually freed. Lisp guarantees that unused space will eventually be reclaimed by the garbage collector. This happens automatically—the programmer need not and indeed can not explicitly free storage. The problem is that the efficiency of garbage collection can vary widely. Garbage collection is particularly worrisome for real-time systems, because it can happen at any time.

The antidote to garbage woes is to avoid unnecessary copying of objects in often-used code. Try using destructive operations, like `nreverse`, `delete`, and `nconc`, rather than their nondestructive counterparts, (like `reverse`, `remove`, and `append`) whenever it is safe to do so. Or use vectors instead of lists, and reuse values rather than creating copies. As usual, this gain in efficiency may lead to errors that can

be difficult to debug. However, the most common kind of unnecessary copying can be eliminated by simple reorganization of your code. Consider the following version of `flatten`, which returns a list of all the atoms in its input, preserving order. Unlike the version in chapter 5, this version returns a single list of atoms, with no embedded lists.

```
(defun flatten (input)
  "Return a flat list of the atoms in the input.
  Ex: (flatten '((a) (b (c) d))) => (a b c d)."
  (cond ((null input) nil)
        ((atom input) (list input))
        (t (append (flatten (first input))
                   (flatten (rest input))))))
```

This definition is quite simple, and it is easy to see that it is correct. However, each call to `append` requires copying the first argument, so this version can cons  $O(n^2)$  cells on an input with  $n$  atoms. The problem with this approach is that it computes the list of atoms in the `first` and `rest` of each subcomponent of the input. But the `first` sublist by itself is not part of the final answer—that’s why we have to call `append`. We could avoid generating garbage by replacing `append` with `nconc`, but even then we would still be wasting time, because `nconc` would have to scan through each sublist to find its end.

The version below makes use of an *accumulator* to keep track of the atoms that have been collected in the `rest`, and to add the atoms in the `first` one at a time with `cons`, rather than building up unnecessary sublists and appending them. This way no garbage is generated, and no subcomponent is traversed more than once.

```
(defun flatten (input &optional accumulator)
  "Return a flat list of the atoms in the input.
  Ex: (flatten '((a) (b (c) d))) => (a b c d)."
  (cond ((null input) accumulator)
        ((atom input) (cons input accumulator))
        (t (flatten (first input)
                   (flatten (rest input) accumulator))))))
```

The version with the accumulator may be a little harder to understand, but it is far more efficient than the original version. Experienced Lisp programmers become quite skilled at replacing calls to `append` with accumulators.

Some of the early Lisp machines had unreliable garbage-collection, so users just turned garbage collection off, used the machine for a few days, and rebooted when they ran out of space. With a large virtual memory system this is a feasible approach, because virtual memory is a cheap resource. The problem is that real memory is still an expensive resource. When each page contains mostly garbage

and only a little live data, the system will spend a lot of time paging data in and out. Compacting garbage-collection algorithms can relocate live data, packing it into a minimum number of pages.

Some garbage-collection algorithms have been optimized to deal particularly well with just this case. If your system has an *ephemeral* or *generational* garbage collector, you need not be so concerned with short-lived objects. Instead, it will be the medium-aged objects that cause problems. The other problem with such systems arises when an object in an old generation is changed to point to an object in a newer generation. This is to be avoided, and it may be that `reverse` is actually faster than `nreverse` in such cases. To decide what works best on your particular system, design some test cases and time them.

As an example of efficient use of storage, here is a version of `pat-match` that eliminates (almost) all consing. The original version of `pat-match`, as used in ELIZA (page 180), used an association list of variable/value pairs to represent the binding list. This version uses two sequences: a sequence of variables and a sequence of values. The sequences are implemented as vectors instead of lists. In general, vectors take half as much space as lists to store the same information, since half of every list is just pointing to the next element.

In this case, the savings are much more substantial than just half. Instead of building up small binding lists for each partial match and adding to them when the match is extended, we will allocate a sufficiently large vector of variables and values just once, and use them over and over for each partial match, and even for each invocation of `pat-match`. To do this, we need to know how many variables we are currently using. We could initialize a counter variable to zero and increment it each time we found a new variable in the pattern. The only difficulty would be when the counter variable exceeds the size of the vector. We could just give up and print an error message, but there are more user-friendly alternatives. For example, we could allocate a larger vector for the variables, copy over the existing ones, and then add in the new one.

It turns out that Common Lisp has a built-in facility to do just this. When a vector is created, it can be given a *fill pointer*. This is a counter variable, but one that is conceptually stored inside the vector. Vectors with fill pointers act like a cross between a vector and a stack. You can push new elements onto the stack with the functions `vector-push` or `vector-push-extend`. The latter will automatically allocate a larger vector and copy over elements if necessary. You can remove elements with `vector-pop`, or you can explicitly look at the fill pointer with `fill-pointer`, or change it with a `setf`. Here are some examples (with `*print-array*` set to `t` so we can see the results):

```
> (setf a (make-array 5 :fill-pointer 0)) ⇒ #()
> (vector-push 1 a) ⇒ 0
```

```

> (vector-push 2 a) ⇒ 1
> a ⇒ #(1 2)
> (vector-pop a) ⇒ 2
> a ⇒ #(1)
> (dotimes (i 10) (vector-push-extend 'x a)) ⇒ NIL
> a ⇒ #(1 X X X X X X X X X X)
> (fill-pointer a) ⇒ 11
> (setf (fill-pointer a) 1) ⇒ 1
> a ⇒ #(1)
> (find 'x a) ⇒ NIL NIL      ; FIND can't find past the fill pointer
> (aref a 2) ⇒ X            ; But AREF can see beyond the fill pointer

```

Using vectors with fill pointers in pat-match, the total storage for binding lists is just twice the number of variables in the largest pattern. I have arbitrarily picked 10 as the maximum number of variables, but even this is not a hard limit, because vector-push-extend can increase it. In any case, the total storage is small, fixed in size, and amortized over all calls to pat-match. These are just the features that indicate a responsible use of storage.

However, there is a grave danger with this approach: the value returned must be managed carefully. The new pat-match returns the value of success when it matches. success is bound to a cons of the variable and value vectors. These can be freely manipulated by the calling routine, but only up until the next call to pat-match. At that time, the contents of the two vectors can change. Therefore, if any calling function needs to hang on to the returned value after another call to pat-match, it should make a copy of the returned value. So it is not quite right to say that this version of pat-match eliminates all consing. It will cons when vector-push-extend runs out of space, or when the user needs to make a copy of a returned value.

Here is the new definition of pat-match. It is implemented by closing the definition of pat-match and its two auxilliary functions inside a let that establishes the bindings of vars, vals, and success, but that is not crucial. Those three variables could have been implemented as global variables instead. Note that it does not support segment variables, or any of the other options implemented in the pat-match of chapter 6.

```
(let* ((vars (make-array 10 :fill-pointer 0 :adjustable t))
      (vals (make-array 10 :fill-pointer 0 :adjustable t))
      (success (cons vars vals)))
```

```
(defun efficient-pat-match (pattern input)
  "Match pattern against input."
  (setf (fill-pointer vars) 0)
  (setf (fill-pointer vals) 0)
  (pat-match-1 pattern input))

(defun pat-match-1 (pattern input)
  (cond ((variable-p pattern) (match-var pattern input))
        ((eql pattern input) success)
        ((and (consp pattern) (consp input))
         (and (pat-match-1 (first pattern) (first input))
              (pat-match-1 (rest pattern) (rest input))))
        (t fail)))

(defun match-var (var input)
  "Match a single variable against input."
  (let ((i (position var vars)))
    (cond ((null i)
           (vector-push-extend var vars)
           (vector-push-extend input vals)
           success)
          ((equal input (aref vals i)) success)
          (t fail))))
```

An example of its use:

```
> (efficient-pat-match '(?x + ?x = ?y . ?z)
                      '(2 + 2 = (3 + 1) is true))
  (#(?X ?Y ?Z) . #(2 (3 + 1) (IS TRUE)))
```

Extensible vectors with fill pointers are convenient, and much more efficient than consing up lists. However, there is some overhead involved in using them, and for those sections of code that must be most efficient, it is best to stick with simple vectors. The following version of `efficient-pat-match` explicitly manages the size of the vectors and explicitly replaces them with new ones when the size is exceeded:

```
(let* ((current-size 0)
       (max-size 1)
       (vars (make-array max-size))
       (vals (make-array max-size))
       (success (cons vars vals)))
  (declare (simple-vector vars vals)
           (fixnum current-size max-size))
```

```

(defun efficient-pat-match (pattern input)
  "Match pattern against input."
  (setf current-size 0)
  (pat-match-1 pattern input))

;; pat-match-1 is unchanged

(defun match-var (var input)
  "Match a single variable against input."
  (let ((i (position var vars)))
    (cond
      ((null i)
       (when (= current-size max-size)
         ;; Make new vectors when we run out of space
         (setf max-size (* 2 max-size)
               vars (replace (make-array max-size) vars)
               vals (replace (make-array max-size) vals)
               success (cons vars vals)))
         ;; Store var and its value in vectors
         (setf (aref vars current-size) var)
         (setf (aref vals current-size) input)
         (incf current-size)
         success)
       ((equal input (aref vals i)) success)
       (t fail)))))


```

In conclusion, replacing lists with vectors can often save garbage. But when you must use lists, it pays to use a version of `cons` that avoids consing when possible. The following is such a version:

```

(proclaim '(inline reuse-cons))

(defun reuse-cons (x y x-y)
  "Return (cons x y), or just x-y if it is equal to (cons x y)."
  (if (and (eql x (car x-y)) (eql y (cdr x-y)))
      x-y
      (cons x y)))

```

The trick is based on the definition of `subst` in Steele's *Common Lisp the Language*. Here is a definition for a version of `remove` that uses `reuse-cons`:

```
(defun remq (item list)
  "Like REMOVE, but uses EQ, and only works on lists."
  (cond ((null list) nil)
        ((eq item (first list)) (remq item (rest list)))
        (t (reuse-cons (first list)
                      (remq item (rest list))
                      list))))
```

### *Avoid Consing: Unique Lists*

Of course, `reuse-cons` only works when you have candidate cons cells around. That is, `(reuse-cons a b c)` only saves space when `c` is (or might be) equal to `(cons a b)`. For some applications, it is useful to have a version of `cons` that returns a unique cons cell without needing `c` as a hint. We will call this version `ucons` for “unique cons.” `ucons` maintains a double hash table: `*uniq-cons-table*` is a hash table whose keys are the cars of cons cells. The value for each car is another hash table whose keys are the cdrs of cons cells. The value of each cdr in this second table is the original cons cell. So two different cons cells with the same car and cdr will retrieve the same value. Here is an implementation of `ucons`:

```
(defvar *uniq-cons-table* (make-hash-table :test #'eq))

(defun ucons (x y)
  "Return a cons s.t. (eq (ucons x y) (ucons x y)) is true."
  (let ((car-table (or (gethash x *uniq-cons-table*)
                       (setf (gethash x *uniq-cons-table*)
                             (make-hash-table :test #'eq)))))
    (or (gethash y car-table)
        (setf (gethash y car-table) (cons x y)))))
```

`ucons`, unlike `cons`, is a true function: it will always return the same value, given the same arguments, where “same” is measured by `eq`. However, if `ucons` is given arguments that are equal but not `eq`, it will not return a unique result. For that we need the function `unique`. It has the property that `(unique x)` is `eq` to `(unique y)` whenever `x` and `y` are equal. `unique` uses a hash table for atoms in addition to the double hash table for conses. This is necessary because strings and arrays can be equal without being `eq`. Besides `unique`, we also define `ulist` and `uappend` for convenience.

```
(defvar *uniq-atom-table* (make-hash-table :test #'equal))
```

```
(defun unique (exp)
  "Return a canonical representation that is EQUAL to exp,
  such that (equal x y) implies (eq (unique x) (unique y))."
  (typecase exp
    (symbol exp)
    (fixnum exp) ; Remove if fixnums are not eq in your Lisp
    (atom   (or (gethash exp *uniq-atom-table*)
                (setf (gethash exp *uniq-atom-table*) exp)))
    (cons   (unique-cons (car exp) (cdr exp)))))

(defun unique-cons (x y)
  "Return a cons s.t. (eq (ucons x y) (ucons x2 y2)) is true
  whenever (equal x x2) and (equal y y2) are true."
  (ucons (unique x) (unique y)))

(defun ulist (&rest args)
  "A uniquified list."
  (unique args))

(defun uappend (x y)
  "A unique list equal to (append x y)."
  (if (null x)
      (unique y)
      (ucons (first x) (uappend (rest x) y)))))
```

The above code works, but it can be improved. The problem is that when `unique` is applied to a tree, it always traverses the tree all the way to the leaves. The function `unique-cons` is like `ucons`, except that `unique-cons` assumes its arguments are not yet unique. We can modify `unique-cons` so that it first checks to see if its arguments are unique, by looking in the appropriate hash tables:

```
(defun unique-cons (x y)
  "Return a cons s.t. (eq (ucons x y) (ucons x2 y2)) is true
  whenever (equal x x2) and (equal y y2) are true."
  (let ((ux) (uy)) ; unique x and y
    (let ((car-table
           (or (gethash x *uniq-cons-table*)
               (gethash (setf ux (unique x)) *uniq-cons-table*)
               (setf (gethash ux *uniq-cons-table*)
                     (make-hash-table :test #'eq)))))
        (or (gethash y car-table)
            (gethash (setf uy (unique y)) car-table)
            (setf (gethash uy car-table)
                  (cons ux uy))))))
```

Another advantage of `unique` is that it can help in indexing. If lists are unique, then they can be stored in an `eq` hash table instead of a `equal` hash table. This can

lead to significant savings when the list structures are large. An eq hash table for lists is almost as good as a property list on symbols.

### *Avoid Consing: Multiple Values*

Parameters and multiple values can also be used to pass around values, rather than building up lists. For example, instead of:

```
(defstruct point "A point in 3-D cartesian space." x y z)

(defun scale-point (k pt)
  "Multiply a point by a constant, K."
  (make-point :x (* k (point-x pt))
              :y (* k (point-y pt))
              :z (* k (point-z pt))))
```

one could use the following approach, which doesn't generate structures:

```
(defun scale-point (k x y z)
  "Multiply the point (x,y,z) by a constant, K."
  (values (* k x) (* k y) (* k z)))
```

### *Avoid Consing: Resources*

Sometimes it pays to manage explicitly the storage of instances of some data type. A pool of these instances may be called a *resource*. Explicit management of a resource is appropriate when: (1) instances are frequently created, and are needed only temporarily; (2) it is easy/possible to be sure when instances are no longer needed; and (3) instances are fairly large structures or take a long time to initialize, so that it is worth reusing them instead of creating new ones. Condition (2) is the crucial one: If you deallocate an instance that is still being used, that instance will mysteriously be altered when it is reallocated. Conversely, if you fail to deallocate unneeded instances, then you are wasting valuable memory space. (The memory management scheme is said to leak in this case.)

The beauty of using Lisp's built-in memory management is that it is guaranteed never to leak and never to deallocate structures that are in use. This eliminates two potential bug sources. The penalty you pay for this guarantee is some inefficiency of the general-purpose memory management as compared to a custom user-supplied management scheme. But beware: modern garbage-collection techniques are highly optimized. In particular, the so-called *generation scavenging* or *ephemeral* garbage collectors look more often at recently allocated storage, on the grounds that recently made objects are more likely to become garbage. If you hold on to garbage in your own data structures, you may end up with worse performance.

With all these warnings in mind, here is some code to manage resources:

```
(defmacro defresource (name &key constructor (initial-copies 0)
                           (size (max initial-copies 10)))
  (let ((resource (symbol name '-resource)))
    (deallocate (symbol 'deallocate- name))
    (allocate (symbol 'allocate- name)))
  '(let ((,resource (make-array ,size :fill-pointer 0)))
    (defun ,allocate ()
      "Get an element from the resource pool, or make one."
      (if (= (fill-pointer ,resource) 0)
          ,constructor
          (vector-pop ,resource)))
    (defun ,deallocate (,name)
      "Place a no-longer-needed element back in the pool."
      (vector-push-extend ,name ,resource))
    ,(if (> initial-copies 0)
        '(mapc #'deallocate (loop repeat ,initial-copies
                                    collect (,allocate))))
        ',name)))
```

Let's say we had some structure called a buffer which we were constantly making instances of and then discarding. Furthermore, suppose that buffers are fairly complex objects to build, that we know we'll need at least 10 of them at a time, and that we probably won't ever need more than 100 at a time. We might use the buffer resource as follows:

```
(defresource buffer :constructor (make-buffer)
            :size 100 :initial-copies 10)
```

This expands into the following code:

```
(let ((buffer-resource (make-array 100 :fill-pointer 0)))
  (defun allocate-buffer ()
    "Get an element from the resource pool, or make one."
    (if (= (fill-pointer buffer-resource) 0)
        (make-buffer)
        (vector-pop buffer-resource)))
  (defun deallocate-buffer (buffer)
    "Place a no-longer-needed element back in the pool."
    (vector-push-extend buffer buffer-resource))
  (mapc #'deallocate-buffer
        (loop repeat 10 collect (allocate-buffer)))
  'buffer)
```

We could then use:

```
(let ((b (allocate-buffer)))
  ...
  (process b)
  ...
  (deallocate-buffer b)))
```

The important thing to remember is that this works only if the buffer *b* really can be deallocated. If the function *process* stored away a pointer to *b* somewhere, then it would be a mistake to deallocate *b*, because a subsequent allocation could unpredictably alter the stored buffer. Of course, if *process* stored a *copy* of *b*, then everything is alright. This pattern of allocation and deallocation is so common that we can provide a macro for it:

```
(defmacro with-resource ((var resource &optional protect) &rest body)
  "Execute body with VAR bound to an instance of RESOURCE."
  (let ((allocate (symbol 'allocate- resource))
        (deallocate (symbol 'deallocate- resource)))
    (if protect
        '(let ((,var nil))
          (unwind-protect
            (progn (setf ,var (,allocate)) ,@body)
            (unless (null ,var) (,deallocate ,var))))
        '(let ((,var (,allocate)))
          ,@body
          (,deallocate ,var)))))
```

The macro allows for an optional argument that sets up an *unwind-protect* environment, so that the buffer gets deallocated even when the body is abnormally exited. The following expansions should make this clearer:

```
> (macroexpand '(with-resource (b buffer)
                                "... (process b) ..."))
(let ((b (allocate-buffer)))
  ...
  (process b)
  ...
  (deallocate-buffer b))

> (macroexpand '(with-resource (b buffer t)
                                "... (process b) ..."))
(let ((b nil))
  (unwind-protect
    (progn (setf b (allocate-buffer))
           ...
           (process b)
           ...
           (deallocate-buffer b))))
```

```
(process b)
  "...")
(unless (null b)
  (deallocate-buffer b))))
```

An alternative to full resources is to just save a single data object. Such an approach is simpler because there is no need to index into a vector of objects, but it is sufficient for some applications, such as a tail-recursive function call that only uses one object at a time.

Another possibility is to make the system slower but safer by having the deallocate function check that its argument is indeed an object of the correct type.

Keep in mind that using resources may put you at odds with the Lisp system's own storage management scheme. In particular, you should be concerned with paging performance on virtual memory systems. A common problem is to have only a few live objects on each page, thus forcing the system to do a lot of paging to get any work done. Compacting garbage collectors can collect live objects onto the same page, but using resources may interfere with this.

## 10.5 Use the Right Data Structures

It is important to implement key data types with the most efficient implementation. This can vary from machine to machine, but there are a few techniques that are universal. Here we consider three case studies.

### *The Right Data Structure: Variables*

As an example, consider the implementation of pattern-matching variables. We saw from the instrumentation of `simplify` that `variable-p` was one of the most frequently used functions. In compiling the matching expressions, I did away with all calls to `variable-p`, but let's suppose we had an application that required run-time use of variables. The specification of the data type `variable` will include two operators, the recognizer `variable-p`, and the constructor `make-variable`, which gives a new, previously unused variable. (This was not needed in the pattern matchers shown so far, but will be needed for unification with backward chaining.) One implementation of variables is as symbols that begin with the character `#\?:`

```
(defun variable-p (x)
  "Is x a variable (a symbol beginning with '?')?"
  (and (symbolp x) (equal (elt (symbol-name x) 0) #\?)))
```

```
(defun make-variable () "Generate a new variable" (gentemp "?"))
```

We could try to speed things up by changing the implementation of variables to be keywords and making the functions inline:

```
(proclaim '(inline variable-p make-variable))
(defun variable-p (x) "Is x a variable?" (keywordp x))
(defun make-variable () (gentemp "X" #.(find-package "KEYWORD")))
```

(The reader character sequence #. means to evaluate at read time, rather than at execution time.) On my machine, this implementation is pretty fast, and I accepted it as a viable compromise. However, other implementations were also considered. One was to have variables as structures, and provide a read macro and print function:

```
(defstruct (variable (:print-function print-variable)) name)
(defvar *vars* (make-hash-table))
(set-macro-character #\?
 #'(lambda (stream char)
    ;; Find an old var, or make a new one with the given name
    (declare (ignore char))
    (let ((name (read stream t nil t)))
      (or (gethash name *vars*)
          (setf (gethash name *vars*) (make-variable :name name)))))

(defun print-variable (var stream depth)
  (declare (ignore depth))
  (format stream "?~a" (var-name var)))
```

It turned out that, on all three Lisps tested, structures were slower than keywords or symbols. Another alternative is to have the ? read macro return a cons whose first is, say, :var. This requires a special output routine to translate back to the ? notation. Yet another alternative, which turned out to be the fastest of all, was to implement variables as negative integers. Of course, this means that the user cannot use negative integers elsewhere in patterns, but that turned out to be acceptable for the application at hand. The moral is to know which features are done well in your particular implementation and to go out of your way to use them in critical situations, but to stick with the most straightforward implementation in noncritical sections.

Lisp makes it easy to rely on lists, but one must avoid the temptation to overuse lists; to use them where another data structure is more appropriate. For example, if you need to access elements of a sequence in arbitrary order, then a vector is more appropriate than list. If the sequence can grow, use an adjustable vector. Consider the problem of maintaining information about a set of people, and searching that set. A naive implementation might look like this:

---

```
(defvar *people* nil "Will hold a list of people")
(defstruct person name address id-number)
(defun person-with-id (id)
  (find id *people* :key #'person-id-number))
```

In a traditional language like C, the natural solution is to include in the person structure a pointer to the next person, and to write a loop to follow these pointers. Of course, we can do that in Lisp too:

```
(defstruct person name address id-number next)
(defun person-with-id (id)
  (loop for person = *people* then (person-next person)
        until (null person)
        do (when (eql id (person-id-number person))
              (RETURN person))))
```

This solution takes less space and is probably faster, because it requires less memory accesses: one for each person rather than one for each person plus one for each cons cell. So there is a small price to pay for using lists. But Lisp programmers feel that price is worth it, because of the convenience and ease of coding and debugging afforded by general-purpose functions like `find`.

In any case, if there are going to be a large number of people, the list is definitely the wrong data structure. Fortunately, Lisp makes it easy to switch to more efficient data structures, for example:

```
(defun person-with-id (id)
  (gethash id *people*))
```

### *The Right Data Structure: Queues*

A *queue* is a data structure where one can add elements at the rear and remove them from the front. This is almost like a stack, except that in a stack, elements are both added and removed at the same end.

Lists can be used to implement stacks, but there is a problem in using lists to implement queues: adding an element to the rear requires traversing the entire list. So collecting  $n$  elements would be  $O(n^2)$  instead of  $O(n)$ .

An alternative implementation of queues is as a cons of two pointers: one to the list of elements of the queue (the contents), and one to the last cons cell in the list. Initially, both pointers would be nil. This implementation in fact existed in BBN Lisp and UCI Lisp under the function name `tconc`:

```
;;; A queue is a (contents . last) pair

(defun tconc (item q)
  "Insert item at the end of the queue."
  (setf (cdr q)
    (if (null (cdr q))
      (setf (car q) (cons item nil))
      (setf (rest (cdr q))
        (cons item nil)))))
```

The `tconc` implementation has the disadvantage that adding the first element to the contents is different from adding subsequent elements, so an `if` statement is required to decide which action to take. The definition of queues given below avoids this disadvantage with a clever trick. First, the order of the two fields is reversed. The `car` of the cons cell is the last element, and the `cdr` is the contents. Second, the empty queue is a cons cell where the `cdr` (the contents field) is `nil`, and the `car` (the last field) is the cons itself. In the definitions below, we change the name `tconc` to the more standard `enqueue`, and provide the other queue functions as well:

```
;;; A queue is a (last . contents) pair

(proclaim '(inline queue-contents make-queue enqueue dequeue
               front empty-queue-p queue-nconc))

(defun queue-contents (q) (cdr q))

(defun make-queue ()
  "Build a new queue, with no elements."
  (let ((q (cons nil nil)))
    (setf (car q) q)))

(defun enqueue (item q)
  "Insert item at the end of the queue."
  (setf (car q)
    (setf (rest (car q))
      (cons item nil)))
  q)

(defun dequeue (q)
  "Remove an item from the front of the queue."
  (pop (cdr q))
  (if (null (cdr q)) (setf (car q) q)
  q))

(defun front (q) (first (queue-contents q)))

(defun empty-queue-p (q) (null (queue-contents q)))
```

---

```
(defun queue-nconc (q list)
  "Add the elements of LIST to the end of the queue."
  (setf (car q)
        (last (setf (rest (car q)) list))))
```

### *The Right Data Structure: Tables*

A *table* is a data structure to which one can insert a key and associate it with a value, and later use the key to look up the value. Tables may have other operations, like counting the number of keys, clearing out all keys, or mapping a function over each key/value pair.

Lisp provides a wide variety of choices to implement tables. An association list is perhaps the simplest: it is just a list of key/value pairs. It is appropriate for small tables, up to a few dozen pairs. The hash table is designed to be efficient for large tables, but may have significant overhead for small ones. If the keys are symbols, property lists can be used. If the keys are integers in a narrow range (or can be mapped into them), then a vector may be the most efficient choice.

Here we implement an alternative data structure, the *trie*. A trie implements a table for keys that are composed of a finite sequence of components. For example, if we were implementing a dictionary as a trie, each key would be a word, and each letter of the word would be a component. The value of the key would be the word's definition. At the top of the dictionary trie is a multiway branch, one for each possible first letter. Each second-level node has a branch for every possible second letter, and so on. To find an  $n$ -letter word requires  $n$  reads. This kind of organization is especially good when the information is stored on secondary storage, because a single read can bring in a node with all its possible branches.

If the keys can be arbitrary list structures, rather than a simple sequence of letters, we need to regularize the keys, transforming them into a simple sequence. One way to do that makes use of the fact that any tree can be written as a linear sequence of atoms and cons operations, in prefix form. Thus, we would make the following transformation:

```
(a (b c) d) ≡
(cons a (cons (cons b (cons c nil)) (cons d nil))) ≡
(cons a  cons  cons b  cons c nil      cons d nil)
```

In the implementation of tries below, this transformation is done on the fly: The four user-level functions are `make-trie` to create a new trie, `put-trie` and `get-trie` to add and retrieve key/value pairs, and `delete-trie` to remove them.

Notice that we use a distinguished value to mark deleted elements, and that `get-trie` returns two values: the actual value found, and a flag saying if anything

was found or not. This is consistent with the interface to gethash and find, and allows us to store null values in the trie. It is an inobtrusive choice, because the programmer who decides not to store null values can just ignore the second value, and everything will work properly.

```
(defstruct trie (value nil) (arcs nil))
(defconstant trie-deleted "deleted")

(defun put-trie (key trie value)
  "Set the value of key in trie."
  (setf (trie-value (find-trie key t trie)) value))

(defun get-trie (key trie)
  "Return the value for a key in a trie, and t/nil if found."
  (let* ((key-trie (find-trie key nil trie)))
    (val (if key-trie (trie-value key-trie))))
  (if (or (null key-trie) (eq val trie-deleted))
      (values nil nil)
      (values val t)))

(defun delete-trie (key trie)
  "Remove a key from a trie."
  (put-trie key trie trie-deleted))

(defun find-trie (key extend? trie)
  "Find the trie node for this key.
  If EXTEND? is true, make a new node if need be."
  (cond ((null trie) nil)
        ((atom key)
         (follow-arc key extend? trie))
        (t (find-trie
             (cdr key) extend?
             (find-trie
              (car key) extend?
              (find-trie
               ." extend? trie))))))

(defun follow-arc (component extend? trie)
  "Find the trie node for this component of the key.
  If EXTEND? is true, make a new node if need be."
  (let ((arc (assoc component (trie-arcs trie))))
    (cond ((not (null arc)) (cdr arc))
          ((not extend?) nil)
          (t (let ((new-trie (make-trie)))
               (push (cons component new-trie)
                     (trie-arcs trie))
               new-trie)))))
```

There are a few subtleties in the implementation. First, we test for deleted entries with an `eq` comparison to a distinguished marker, the string `trie-deleted`. No other object will be `eq` to this string except `trie-deleted` itself, so this is a good test. We also use a distinguished marker, the string `". "`, to mark cons cells. Components are implicitly compared against this marker with an `eql` test by the `assoc` in `follow-arc`. Maintaining the identity of this string is crucial; if, for example, you recompiled the definition of `find-trie` (without changing the definition at all), then you could no longer find keys that were indexed in an existing trie, because the `". "` used by `find-trie` would be a different one from the `". "` in the existing trie.

*Artificial Intelligence Programming* (Charniak et al. 1987) discusses variations on the trie, particularly in the indexing scheme. If we always use proper lists (no non-null cdrs), then a more efficient encoding is possible. As usual, the best type of indexing depends on the data to be indexed. It should be noted that Charniak et al. call the trie a *discrimination net*. In general, that term refers to any tree with tests at the nodes.

A trie is, of course, a kind of tree, but there are cases where it pays to convert a trie into a *dag*—a directed acyclic graph. A dag is a tree where some of the subtrees are shared. Imagine you have a spelling corrector program with a list of some 50,000 or so words. You could put them into a trie, each word with the value `t`. But there would be many subtrees repeated in this trie. For example, given a word list containing `look`, `looks`, `looked`, and `looking` as well as `show`, `shows`, `showed`, and `showing`, there would be repetition of the subtree containing `-s`, `-ed` and `-ing`. After the trie is built, we could pass the whole trie to `unique`, and it would collapse the shared subtrees, saving storage. Of course, you can no longer add or delete keys from the dag without risking unintended side effects.

This process was carried out for a 56,000 word list. The trie took up 3.2Mbytes, while the dag was 1.1Mbytes. This was still deemed unacceptable, so a more compact encoding of the dag was created, using a .2Mbytes vector. Encoding the same word list in a hash table took twice this space, even with a special format for encoding suffixes.

Tries work best when neither the indexing key nor the retrieval key contains variables. They work reasonably well when the variables are near the end of the sequence. Consider looking up the pattern `"yello?"` in the dictionary, where the `"?"` character indicates a match of any letter. Following the branches for `"yello"` leads quickly to the only possible match, `"yellow"`. In contrast, fetching with the pattern `"??llo"` is much less efficient. The table lookup function would have to search all 26 top-level branches, and for each of those consider all possible second letters, and for each of those consider the path `"llow"`. Quite a bit of searching is required before arriving at the complete set of matches: `bellow`, `billow`, `fallow`, `fellow`, `follow`, `hallow`, `hollow`, `mallow`, `mellow`, `pillow`, `sallow`, `tallow`, `wallow`, `willow`, and `yellow`.

We will return to the problem of discrimination nets with variables in section 14.8, page 472.

## 10.6 Exercises

 **Exercise 10.1 [h]** Define the macro `deftable`, such that `(deftable person assoc)` will act much like a `defstruct`—it will define a set of functions for manipulating a table of people: `get-person`, `put-person`, `clear-person`, and `map-person`. The table should be implemented as an association list. Later on, you can change the representation of the table simply by changing the form to `(deftable person hash)`, without having to change anything else in your code. Other implementation options include property lists and vectors. `deftable` should also take three keyword arguments: `inline`, `size` and `test`. Here is a possible macroexpansion:

```
> (macroexpand '(deftable person hash :inline t :size 100)) ≡
  (progn
    (proclaim '(inline get-person put-person map-person))
    (defparameter *person-table*
      (make-hash-table :test #'eql :size 100))
    (defun get-person (x &optional default)
      (gethash x *person-table* default))
    (defun put-person (x value)
      (setf (gethash x *person-table*) value))
    (defun clear-person () (clrhash *person-table*))
    (defun map-person (fn) (maphash fn *person-table*))
    (defsetf get-person put-person)
  'person)
```

 **Exercise 10.2 [m]** We can use the `:type` option to `defstruct` to define structures implemented as lists. However, often we have a two-field structure that we would like to implement as a cons cell rather than a two-element list, thereby cutting storage in half. Since `defstruct` does not allow this, define a new macro that does.

 **Exercise 10.3 [m]** Use `reuse-cons` to write a version of `flatten` (see page 329) that shares as much of its input with its output as possible.

 **Exercise 10.4 [h]** Consider the data type *set*. A set has two main operations: `adjoin` an element and test for membership. It is convenient to also add a map-over-elements operation. With these primitive operations it is possible to build up more complex operations like `union` and `intersection`.

As mentioned in section 3.9, Common Lisp provides several implementations of sets. The simplest uses lists as the underlying representation, and provides the

functions `adjoin`, `member`, `union`, `intersection`, and `set-difference`. Another uses bit vectors, and a similar one uses integers viewed as bit sequences. Analyze the time complexity of each implementation for each operation.

Next, show how *sorted lists* can be used to implement sets, and compare the operations on sorted lists to their counterparts on unsorted lists.

## 10.7 Answers

### Answer 10.2

```
(defmacro def-cons-struct (cons car cdr &optional inline?)
  "Define aliases for cons, car and cdr."
  '(progn (proclaim ',(if inline? 'inline 'notinline)
                     ,car ,cdr ,cons))
         (defun ,car (x) (car x))
         (defun ,cdr (x) (cdr x))
         (defsetf ,car (x) (val) '(setf (car ,x) ,val))
         (defsetf ,cdr (x) (val) '(setf (cdr ,x) ,val))
         (defun ,cons (x y) (cons x y))))
```

### Answer 10.3

```
(defun flatten (exp &optional (so-far nil) last-cons)
  "Return a flat list of the atoms in the input.
  Ex: (flatten '((a) (b (c) d))) => (a b c d)."
  (cond ((null exp) so-far)
        ((atom exp) (reuse-cons exp so-far last-cons))
        (t (flatten (first exp)
                    (flatten (rest exp) so-far exp)
                    exp))))
```

## CHAPTER 11

---

# Logic Programming

*A language that doesn't affect the way you think  
about programming is not worth knowing.*

—Alan Perlis

Lisp is the major language for AI work, but it is by no means the only one. The other strong contender is Prolog, whose name derives from “programming in logic.”<sup>1</sup> The idea behind logic programming is that the programmer should state the relationships that describe a problem and its solution. These relationships act as constraints on the algorithms that can solve the problem, but the system itself, rather than the programmer, is responsible for the details of the algorithm. The tension between the “programming” and “logic” will be covered in chapter 14, but for now it is safe to say that Prolog is an approximation to the ideal goal of logic programming. Prolog has arrived at a comfortable niche between a traditional programming language and a logical specification language. It relies on three important ideas:

---

<sup>1</sup>Actually, *programmation en logique*, since it was invented by a French group (see page 382).

- Prolog encourages the use of a single *uniform data base*. Good compilers provide efficient access to this data base, reducing the need for vectors, hash tables, property lists, and other data structures that the Lisp programmer must deal with in detail. Because it is based on the idea of a data base, Prolog is *relational*, while Lisp (and most languages) are *functional*. In Prolog we would represent a fact like “the population of San Francisco is 750,000” as a relation. In Lisp, we would be inclined to write a function, *population*, which takes a city as input and returns a number. Relations are more flexible; they can be used not only to find the population of San Francisco but also, say, to find the cities with populations over 500,000.
- Prolog provides *logic variables* instead of “normal” variables. A logic variable is bound by *unification* rather than by assignment. Once bound, a logic variable can never change. Thus, they are more like the variables of mathematics. The existence of logic variables and unification allow the logic programmer to state equations that constrain the problem (as in mathematics), without having to state an order of evaluation (as with assignment statements).
- Prolog provides *automatic backtracking*. In Lisp each function call returns a single value (unless the programmer makes special arrangements to have it return multiple values, or a list of values). In Prolog, each query leads to a search for relations in the data base that satisfy the query. If there are several, they are considered one at a time. If a query involves multiple relations, as in “what city has a population over 500,000 and is a state capital?,” Prolog will go through the *population* relation to find a city with a population over 500,000. For each one it finds, it then checks the *capital* relation to see if the city is a capital. If it is, Prolog prints the city; otherwise it *backtracks*, trying to find another city in the *population* relation. So Prolog frees the programmer from worrying about both how data is stored and how it is searched. For some problems, the naive automatic search will be too inefficient, and the programmer will have to restate the problem. But the ideal is that Prolog programs state constraints on the solution, without spelling out in detail how the solutions are achieved.

This chapter serves two purposes: it alerts the reader to the possibility of writing certain programs in Prolog rather than Lisp, and it presents implementations of the three important Prolog ideas, so that they may be used (independently or together) within Lisp programs. Prolog represents an interesting, different way of looking at the programming process. For that reason it is worth knowing. In subsequent chapters we will see several useful applications of the Prolog approach.

## 11.1 Idea 1: A Uniform Data Base

The first important Prolog idea should be familiar to readers of this book: manipulating a stored data base of assertions. In Prolog the assertions are called *clauses*, and they can be divided into two types: *facts*, which state a relationship that holds between some objects, and *rules*, which are used to state contingent facts. Here are representations of two facts about the population of San Francisco and the capital of California. The relations are *population* and *capital*, and the objects that participate in these relations are SF, 750000, Sacramento, and CA:

```
(population SF 750000)
(capital Sacramento CA)
```

We are using Lisp syntax, because we want a Prolog interpreter that can be imbedded in Lisp. The actual Prolog notation would be *population(sf,750000)*. Here are some facts pertaining to the *likes* relation:

```
(likes Kim Robin)
(likes Sandy Lee)
(likes Sandy Kim)
(likes Robin cats)
```

These facts could be interpreted as meaning that Kim likes Robin, Sandy likes both Lee and Kim, and Robin likes cats. We need some way of telling Lisp that these are to be interpreted as Prolog facts, not a Lisp function call. We will use the macro *<-* to mark facts. Think of this as an assignment arrow which adds a fact to the data base:

```
(<- (likes Kim Robin))
(<- (likes Sandy Lee))
(<- (likes Sandy Kim))
(<- (likes Robin cats))
```

One of the major differences between Prolog and Lisp hinges on the difference between relations and functions. In Lisp, we would define a function *likes*, so that *(likes 'Sandy)* would return the list *(Lee Kim)*. If we wanted to access the information the other way, we would define another function, say, *likers-of*, so that *(likers-of 'Lee)* returns *(Sandy)*. In Prolog, we have a single *likes* relation instead of multiple functions. This single relation can be used as if it were multiple functions by posing different queries. For example, the query *(likes Sandy ?who)* succeeds with *?who* bound to Lee or Kim, and the query *(likes ?who Lee)* succeeds with *?who* bound to Sandy.

The second type of clause in a Prolog data base is the *rule*. Rules state contingent facts. For example, we can represent the rule that Sandy likes anyone who likes cats as follows:

```
(<- (likes Sandy ?x) (likes ?x cats))
```

This can be read in two ways. Viewed as a logical assertion, it is read, "For any  $x$ , Sandy likes  $x$  if  $x$  likes cats." This is a *declarative* interpretation. Viewed as a piece of a Prolog program, it is read, "If you ever want to show that Sandy likes some  $x$ , one way to do it is to show that  $x$  likes cats." This is a *procedural* interpretation. It is called a *backward-chaining* interpretation, because one reasons backward from the goal (Sandy likes  $x$ ) to the premises ( $x$  likes cats). The symbol  $<-$  is appropriate for both interpretations: it is an arrow indicating logical implication, and it points backwards to indicate backward chaining.

It is possible to give more than one procedural interpretation to a declarative form. (We did that in chapter 1, where grammar rules were used to generate both strings of words and parse trees.) The rule above could have been interpreted procedurally as "If you ever find out that some  $x$  likes cats, then conclude that Sandy likes  $x$ ." This would be *forward chaining*: reasoning from a premise to a conclusion. It turns out that Prolog does backward chaining exclusively. Many expert systems use forward chaining exclusively, and some systems use a mixture of the two.

The leftmost expression in a clause is called the *head*, and the remaining ones are called the *body*. In this view, a fact is just a rule that has no body; that is, a fact is true no matter what. In general, then, the form of a clause is:

```
(<- head body...)
```

A clause asserts that the head is true only if all the goals in the body are true. For example, the following clause says that Kim likes anyone who likes both Lee and Kim:

```
(<- (likes Kim ?x) (likes ?x Lee) (likes ?x Kim))
```

This can be read as:

*For any  $x$ , deduce that Kim likes  $x$   
if it can be proved that  $x$  likes Lee and  $x$  likes Kim.*

## 11.2 Idea 2: Unification of Logic Variables

Unification is a straightforward extension of the idea of pattern matching. The pattern-matching functions we have seen so far have always matched a pattern (an expression containing variables) against a constant expression (one with no variables). In unification, two patterns, each of which can contain variables, are matched against each other. Here's an example of the difference between pattern matching and unification:

```
> (pat-match '(?x + ?y) '(2 + 1)) => ((?Y . 1) (?X . 2))
> (unify '(?x + 1) '(2 + ?y)) => ((?Y . 1) (?X . 2))
```

Within the unification framework, variables (such as  $?x$  and  $?y$  above) are called *logic variables*. Like normal variables, a logic variable can be assigned a value, or it can be unbound. The difference is that a logic variable can never be altered. Once it is assigned a value, it keeps that value. Any attempt to unify it with a different value leads to failure. It is possible to unify a variable with the same value more than once, just as it was possible to do a pattern match of  $(?x + ?x)$  with  $(2 + 2)$ .

The difference between simple pattern matching and unification is that unification allows two variables to be matched against each other. The two variables remain unbound, but they become equivalent. If either variable is subsequently bound to a value, then both variables adopt that value. The following example equates the variables  $?x$  and  $?y$  by binding  $?x$  to  $?y$ :

```
> (unify '(f ?x) '(f ?y)) => ((?X . ?Y))
```

Unification can be used to do some sophisticated reasoning. For example, if we have two equations,  $a + a = 0$  and  $x + y = y$ , and if we know that these two equations unify, then we can conclude that  $a$ ,  $x$ , and  $y$  are all 0. The version of unify we will define shows this result by binding  $?y$  to 0,  $?x$  to  $?y$ , and  $?a$  to  $?x$ . We will also define the function unifier, which shows the structure that results from unifying two structures.

```
> (unify '(?a + ?a = 0) '(?x + ?y = ?y)) =>
((?Y . 0) (?X . ?Y) (?A . ?X))
> (unifier '(?a + ?a = 0) '(?x + ?y = ?y)) => (0 + 0 = 0)
```

To avoid getting carried away by the power of unification, it is a good idea to take stock of exactly what unification provides. It *does* provide a way of stating that variables are equal to other variables or expressions. It does *not* provide a way of automatically solving equations or applying constraints other than equality. The following example

makes it clear that unification treats the symbol `+` only as an uninterpreted atom, not as the addition operator:

```
> (unifier '(?a + ?a = 2) '(?x + ?y = ?y)) => (2 + 2 = 2)
```

Before developing the code for `unify`, we repeat here the code taken from the pattern-matching utility (chapter 6):

```
(defconstant fail nil "Indicates pat-match failure")
(defconstant no-bindings '((t . t))
  "Indicates pat-match success, with no variables.")
(defun variable-p (x)
  "Is x a variable (a symbol beginning with '?')?"
  (and (symbolp x) (equal (char (symbol-name x) 0) #\?)))
(defun get-binding (var bindings)
  "Find a (variable . value) pair in a binding list."
  (assoc var bindings))
(defun binding-val (binding)
  "Get the value part of a single binding."
  (cdr binding))
(defun lookup (var bindings)
  "Get the value part (for var) from a binding list."
  (binding-val (get-binding var bindings)))
(defun extend-bindings (var val bindings)
  "Add a (var . value) pair to a binding list."
  (cons (cons var val)
    ;: Once we add a "real" binding,
    ;: we can get rid of the dummy no-bindings
    (if (and (eq bindings no-bindings))
      nil
      bindings)))
(defun match-variable (var input bindings)
  "Does VAR match input?  Uses (or updates) and returns bindings."
  (let ((binding (get-binding var bindings)))
    (cond ((not binding) (extend-bindings var input bindings))
      ((equal input (binding-val binding)) bindings)
      (t fail))))
```

The `unify` function follows; it is identical to `pat-match` (as defined on page 180) except for the addition of the line marked **\*\*\***. The function `unify-variable` also follows `match-variable` closely:

```
(defun unify (x y &optional (bindings no-bindings))
  "See if x and y match with given bindings."
  (cond ((eq bindings fail) fail)
        ((variable-p x) (unify-variable x y bindings))
        ((variable-p y) (unify-variable y x bindings)) ;***
        ((eql x y) bindings)
        ((and (consp x) (consp y))
         (unify (rest x) (rest y)
                (unify (first x) (first y) bindings)))
        (t fail)))

(defun unify-variable (var x bindings)
  "Unify var with x, using (and maybe extending) bindings."
  ;; Warning - buggy version
  (if (get-binding var bindings)
      (unify (lookup var bindings) x bindings)
      (extend-bindings var x bindings)))
```

Unfortunately, this definition is not quite right. It handles simple examples:

```
> (unify '(?x + 1) '(2 + ?y)) => ((?Y . 1) (?X . 2))
> (unify '?x '?y) => ((?X . ?Y))
> (unify '(%x ?x) '(%y ?y)) => ((?Y . ?Y) (?X . ?Y))
```

but there are several pathological cases that it can't contend with:

```
> (unify '(?x ?x ?x) '(%y ?y ?y))
>>Trap #o43622 (PDL-OVERFLOW REGULAR)
The regular push-down list has overflowed.
While in the function GET-BINDING ⇐ UNIFY-VARIABLE ⇐ UNIFY
```

The problem here is that once ?y gets bound to itself, the call to unify inside unify-variable leads to an infinite loop. But matching ?y against itself must always succeed, so we can move the equality test in unify before the variable test. This assumes that equal variables are eql, a valid assumption for variables implemented as symbols (but be careful if you ever decide to implement variables some other way).

```
(defun unify (x y &optional (bindings no-bindings))
  "See if x and y match with given bindings."
  (cond ((eq bindings fail) fail)
        ((eql x y) bindings) ;** moved this line
        ((variable-p x) (unify-variable x y bindings))
        ((variable-p y) (unify-variable y x bindings))
        ((and (consp x) (consp y))
         (unify (rest x) (rest y)
```

---

```
(unify (first x) (first y) bindings))
(t fail)))
```

Here are some test cases:

```
> (unify '(?x ?x) '(?y ?y)) => ((?X . ?Y))
> (unify '(?x ?x ?x) '(?y ?y ?y)) => ((?X . ?Y))
> (unify '(?x ?y) '(?y ?x)) => ((?Y . ?X) (?X . ?Y))
> (unify '(?x ?y a) '(?y ?x ?x))
>>Trap #o43622 (PDL-OVERFLOW REGULAR)
The regular push-down list has overflowed.
While in the function GET-BINDING ⇐ UNIFY-VARIABLE ⇐ UNIFY
```

We have pushed off the problem but not solved it. Allowing both ( $?Y . ?X$ ) and ( $?X . ?Y$ ) in the same binding list is as bad as allowing ( $?Y . ?Y$ ). To avoid the problem, the policy should be never to deal with bound variables, but rather with their values, as specified in the binding list. The function unify-variable fails to implement this policy. It does have a check that gets the binding for var when it is a bound variable, but it should also have a check that gets the value of x, when x is a bound variable:

```
(defun unify-variable (var x bindings)
  "Unify var with x, using (and maybe extending) bindings."
  (cond ((get-binding var bindings)
          (unify (lookup var bindings) x bindings))
        ((and (variable-p x) (get-binding x bindings)) ;***
         (unify var (lookup x bindings) bindings)) ;***
        (t (extend-bindings var x bindings))))
```

Here are some more test cases:

```
> (unify '(?x ?y) '(?y ?x)) => ((?X . ?Y))
> (unify '(?x ?y a) '(?y ?x ?x)) => ((?Y . A) (?X . ?Y))
```

It seems the problem is solved. Now let's try a new problem:

```
> (unify '?x '(f ?x)) => ((?X F ?X))
```

Here  $((?X F ?X))$  really means  $((?X . ((F ?X))))$ , so  $?X$  is bound to  $(F ?X)$ . This represents a circular, infinite unification. Some versions of Prolog, notably Prolog II (Giannesini et al. 1986), provide an interpretation for such structures, but it is tricky to define the semantics of infinite structures.

The easiest way to deal with such infinite structures is just to ban them. This ban can be realized by modifying the unifier so that it fails whenever there is an attempt to unify a variable with a structure containing that variable. This is known in unification circles as the *occurs check*. In practice the problem rarely shows up, and since it can add a lot of computational complexity, most Prolog systems have ignored the occurs check. This means that these systems can potentially produce unsound answers. In the final version of unify following, a variable is provided to allow the user to turn occurs checking on or off.

```
(defparameter *occurs-check* t "Should we do the occurs check?")

(defun unify (x y &optional (bindings no-bindings))
  "See if x and y match with given bindings."
  (cond ((eq bindings fail) fail)
        ((eql x y) bindings)
        ((variable-p x) (unify-variable x y bindings))
        ((variable-p y) (unify-variable y x bindings))
        ((and (consp x) (consp y))
         (unify (rest x) (rest y)
                (unify (first x) (first y) bindings)))
        (t fail)))

(defun unify-variable (var x bindings)
  "Unify var with x, using (and maybe extending) bindings."
  (cond ((get-binding var bindings)
          (unify (lookup var bindings) x bindings))
        ((and (variable-p x) (get-binding x bindings))
         (unify var (lookup x bindings) bindings))
        ((and *occurs-check* (occurs-check var x bindings))
         fail)
        (t (extend-bindings var x bindings)))))

(defun occurs-check (var x bindings)
  "Does var occur anywhere inside x?"
  (cond ((eq var x) t)
        ((and (variable-p x) (get-binding x bindings))
         (occurs-check var (lookup x bindings) bindings))
        ((consp x) (or (occurs-check var (first x) bindings)
                      (occurs-check var (rest x) bindings)))
        (t nil)))
```

Now we consider how unify will be used. In particular, one thing we want is a function for substituting a binding list into an expression. We originally chose association lists as the implementation of bindings because of the availability of the function *sublis*. Ironically, *sublis* won't work any more, because variables can be bound to other variables, which are in turn bound to expressions. The function *subst-bindings* acts like *sublis*, except that it substitutes recursive bindings.

```
(defun subst-bindings (bindings x)
  "Substitute the value of variables in bindings into x,
  taking recursively bound variables into account."
  (cond ((eq bindings fail) fail)
        ((eq bindings no-bindings) x)
        ((and (variable-p x) (get-binding x bindings))
         (subst-bindings bindings (lookup x bindings)))
        ((atom x) x)
        (t (reuse-cons (subst-bindings bindings (car x))
                      (subst-bindings bindings (cdr x))
                      x))))
```

Now let's try unify on some examples:

```
> (unify '(?x ?y a) '(?y ?x ?x)) => ((?Y . A) (?X . ?Y))
> (unify '?x '(f ?x)) => NIL
> (unify '(?x ?y) '((f ?y) (f ?x))) => NIL
> (unify '(?x ?y ?z) '((?y ?z) (?x ?z) (?x ?y))) => NIL
> (unify 'a 'a) => ((T . T))
```

Finally, the function unifier calls unify and substitutes the resulting binding list into one of the arguments. The choice of x is arbitrary; an equal result would come from substituting the binding list into y.

```
(defun unifier (x y)
  "Return something that unifies with both x and y (or fail)."
  (subst-bindings (unify x y) x))
```

Here are some examples of unifier:

```
> (unifier '(?x ?y a) '(?y ?x ?x)) => (A A A)
> (unifier '(((?a * ?x ^ 2) + (?b * ?x) + ?c)
             '(* ?z + (4 * 5) + 3)) =>
((?A * 5 ^ 2) + (4 * 5) + 3)
```

When \*occurs-check\* is false, we get the following answers:

```
> (unify '?x '(f ?x)) => ((?X F ?X))

> (unify '(%x ?y) '((f ?y) (f ?x))) =>
((?Y F ?X) (?X F ?Y))

> (unify '(%x ?y ?z) '(((y ?z) (%x ?z) (%x ?y))) =>
((?Z ?X ?Y) (?Y ?X ?Z) (?X ?Y ?Z))
```

### *Programming with Prolog*

The amazing thing about Prolog clauses is that they can be used to express relations that we would normally think of as “programs,” not “data.” For example, we can define the `member` relation, which holds between an item and a list that contains that item. More precisely, an item is a member of a list if it is either the first element of the list or a member of the rest of the list. This definition can be translated into Prolog almost verbatim:

```
(<- (member ?item (?item . ?rest)))
(<- (member ?item (?x . ?rest)) (member ?item ?rest))
```

Of course, we can write a similar definition in Lisp. The most visible difference is that Prolog allows us to put patterns in the head of a clause, so we don’t need recognizers like `consp` or accessors like `first` and `rest`. Otherwise, the Lisp definition is similar:<sup>2</sup>

```
(defun lisp-member (item list)
  (and (consp list)
        (or (eql item (first list))
            (lisp-member item (rest list)))))
```

If we wrote the Prolog code without taking advantage of the pattern feature, it would look more like the Lisp version:

```
(<- (member ?item ?list)
      (= ?list (?item . ?rest)))
```

---

<sup>2</sup>Actually, this is more like the Lisp `find` than the Lisp `member`. In this chapter we have adopted the traditional Prolog definition of `member`.

---

```
(<- (member ?item ?list)
      (= ?list (?x . ?rest))
      (member ?item ?rest))
```

If we define or in Prolog, we would write a version that is clearly just a syntactic variant of the Lisp version.

```
(<- (member ?item ?list)
      (= ?list (?first . ?rest))
      (or (= ?item ?first)
          (member ?item ?rest)))
```

Let's see how the Prolog version of `member` works. Imagine that we have a Prolog interpreter that can be given a query using the macro `?-`, and that the definition of `member` has been entered. Then we would see:

```
> (?- (member 2 (1 2 3)))
Yes;
> (?- (member 2 (1 2 3 2 1)))
Yes;
Yes;
```

The answer to the first query is "yes" because 2 is a member of the rest of the list. In the second query the answer is "yes" twice, because 2 appears in the list twice. This is a little surprising to Lisp programmers, but there still seems to be a fairly close correspondence between Prolog's and Lisp's `member`. However, there are things that the Prolog `member` can do that Lisp cannot:

```
> (?- (member ?x (1 2 3)))
?X = 1;
?X = 2;
?X = 3;
```

Here `member` is used not as a predicate but as a generator of elements in a list. While Lisp functions always map from a specified input (or inputs) to a specified output, Prolog relations can be used in several ways. For `member`, we see that the first argument, `?x`, can be either an input or an output, depending on the goal that is specified. This power to use a single specification as a function going in several different directions is a very flexible feature of Prolog. (Unfortunately, while it works very well for simple relations like `member`, in practice it does not work well for large programs. It is very difficult to, say, design a compiler and automatically have it work as a disassembler as well.)

Now we turn to the implementation of the Prolog interpreter, as summarized in figure 11.1. The first implementation choice is the representation of rules and facts. We will build a single uniform data base of clauses, without distinguishing rules from facts. The simplest representation of clauses is as a cons cell holding the head and the body. For facts, the body will be empty.

```
;; Clauses are represented as (head . body) cons cells
(defun clause-head (clause) (first clause))
(defun clause-body (clause) (rest clause))
```

The next question is how to index the clauses. Recall the procedural interpretation of a clause: when we want to prove the head, we can do it by proving the body. This suggests that clauses should be indexed in terms of their heads. Each clause will be stored on the property list of the predicate of the head of the clause. Since the data base is now distributed across the property list of various symbols, we represent the entire data base as a list of symbols stored as the value of \*db-predicates\*.

```
;; Clauses are stored on the predicate's plist
(defun get-clauses (pred) (get pred 'clauses))
(defun predicate (relation) (first relation))

(defvar *db-predicates* nil
  "A list of all predicates stored in the database.")
```

Now we need a way of adding a new clause. The work is split up into the macro <-, which provides the user interface, and a function, add-clause, that does the work. It is worth defining a macro to add clauses because in effect we are defining a new language: Prolog-In-Lisp. This language has only two syntactic constructs: the <- macro to add clauses, and the ?- macro to make queries.

```
(defmacro <- (&rest clause)
  "Add a clause to the data base."
  '(add-clause ',clause))

(defun add-clause (clause)
  "Add a clause to the data base, indexed by head's predicate."
  ;;= The predicate must be a non-variable symbol.
  (let ((pred (predicate (clause-head clause))))
    (assert (and (symbolp pred) (not (variable-p pred))))
    (pushnew pred *db-predicates*)
    (setf (get pred 'clauses)
          (nconc (get-clauses pred) (list clause)))
    pred))
```

Now all we need is a way to remove clauses, and the data base will be complete.

<b>Top-Level Macros</b>	
<-	Add a clause to the data base.
?-	Prove a query and print answer(s).
<b>Special Variables</b>	
*db-predicates*	A list of all predicates.
*occurs-check*	Should we check for circular unifications?
<b>Data Types</b>	
clause	Consists of a head and a body.
variable	A symbol starting with a ?.
<b>Major Functions</b>	
add-clause	Add a clause to the data base.
prove	Return a list of possible solutions to goal.
prove-all	Return a list of solutions to the conjunction of goals.
top-level-prove	Prove the goals, and print variables readably.
<b>Auxiliary Functions</b>	
get-clauses	Find all the clauses for a predicate.
predicate	Pick out the predicate from a relation.
clear-db	Remove all clauses (for all predicates) from the data base.
clear-predicate	Remove the clauses for a single predicate.
rename-variables	Replace all variables in x with new ones.
unique-find-anywhere-if	Find all unique leaves satisfying predicate.
show-prolog-solutions	Print the variables in each of the solutions.
show-prolog-vars	Print each variable with its binding.
variables-in	Return a list of all the variables in an expression.
<b>Previously Defined Constants</b>	
fail	An indication that unification has failed.
no-bindings	A successful unification with no variables.
<b>Previously Defined Functions</b>	
unify	Return bindings that unify two expressions (section 11.2).
unify-variable	Unify a variable against an expression.
occurs-check	See if a particular variable occurs inside an expression.
subst-bindings	Substitute bindings into an expression.
get-binding	Get the ( <i>var . val</i> ) binding for a variable.
lookup	Get the value for a variable.
extend-bindings	Add a new variable/value pair to a binding list.
variable-p	Is the argument a variable?
reuse-cons	Like cons, except will reuse an old value if possible.

Figure 11.1: Glossary for the Prolog Interpreter

```
(defun clear-db ()
  "Remove all clauses (for all predicates) from the data base."
  (mapc #'clear-predicate *db-predicates))

(defun clear-predicate (predicate)
  "Remove the clauses for a single predicate."
  (setf (get predicate 'clauses) nil))
```

A data base is useless without a way of getting data out, as well as putting it in. The function prove will be used to prove that a given goal either matches a fact that is in the data base directly or can be derived from the rules. To prove a goal, first find all the candidate clauses for that goal. For each candidate, check if the goal unifies with the head of the clause. If it does, try to prove all the goals in the body of the clause. For facts, there will be no goals in the body, so success will be immediate. For rules, the goals in the body need to be proved one at a time, making sure that bindings from the previous step are maintained. The implementation is straightforward:

```
(defun prove (goal bindings)
  "Return a list of possible solutions to goal."
  (mapcan #'(lambda (clause)
    (let ((new-clause (rename-variables clause)))
      (prove-all (clause-body new-clause)
        (unify goal (clause-head new-clause) bindings))))
    (get-clauses (predicate goal)))))

(defun prove-all (goals bindings)
  "Return a list of solutions to the conjunction of goals."
  (cond ((eq bindings fail) fail)
    ((null goals) (list bindings))
    (t (mapcan #'(lambda (goall-solution)
      (prove-all (rest goals) goall-solution)
      (prove (first goals) bindings))))))
```

The tricky part is that we need some way of distinguishing a variable  $?x$  in one clause from another variable  $?x$  in another clause. Otherwise, a variable used in two different clauses in the course of a proof would have to take on the same value in each clause, which would be a mistake. Just as arguments to a function can have different values in different recursive calls to the function, so the variables in a clause are allowed to take on different values in different recursive uses. The easiest way to keep variables distinct is just to rename all variables in each clause before it is used. The function rename-variables does this.<sup>3</sup>

---

<sup>3</sup>See exercise 11.12 for an alternative approach.

---

```
(defun rename-variables (x)
  "Replace all variables in x with new ones."
  (sublis (mapcar #'(lambda (var) (cons var (gensym (string var))))
                    (variables-in x)))
          x))
```

`Rename-variables` makes use of `gensym`, a function that generates a new symbol each time it is called. The symbol is not interned in any package, which means that there is no danger of a programmer typing a symbol of the same name. The predicate `variables-in` and its auxiliary function are defined here:

```
(defun variables-in (exp)
  "Return a list of all the variables in EXP."
  (unique-find-anywhere-if #'variable-p exp))

(defun unique-find-anywhere-if (predicate tree
                                    &optional found-so-far)
  "Return a list of leaves of tree satisfying predicate,
  with duplicates removed."
  (if (atom tree)
      (if (funcall predicate tree)
          (adjoin tree found-so-far)
          found-so-far)
      (unique-find-anywhere-if
       predicate
       (first tree)
       (unique-find-anywhere-if predicate (rest tree)
                                 found-so-far))))
```

Finally, we need a nice interface to the proving functions. We will use `?-` as a macro to introduce a query. The query might as well allow a conjunction of goals, so `?-` will call `prove-all`. Together, `<-` and `?-` define the complete syntax of our Prolog-In-Lisp language.

```
(defmacro ?- (&rest goals) `'(prove-all ',goals no-bindings))
```

Now we can enter all the clauses given in the prior example:

```
(<- (likes Kim Robin))
(<- (likes Sandy Lee))
(<- (likes Sandy Kim))
(<- (likes Robin cats))
(<- (likes Sandy ?x) (likes ?x cats))
(<- (likes Kim ?x) (likes ?x Lee) (likes ?x Kim))
(<- (likes ?x ?x))
```

To ask whom Sandy likes, we would use:

```
> (?- (likes Sandy ?who))
(((?WHO . LEE))
 ((?WHO . KIM))
 ((?X2856 . ROBIN) (?WHO . ?X2856))
 ((?X2860 . CATS) (?X2857 . CATS) (?X2856 . SANDY) (?WHO . ?X2856))
 ((?X2865 . CATS) (?X2856 . ?X2865) (?WHO . ?X2856))
 ((?WHO . SANDY) (?X2867 . SANDY)))
```

Perhaps surprisingly, there are six answers. The first two answers are Lee and Kim, because of the facts. The next three stem from the clause that Sandy likes everyone who likes cats. First, Robin is an answer because of the fact that Robin likes cats. To see that Robin is the answer, we have to unravel the bindings: ?who is bound to ?x2856, which is in turn bound to Robin.

Now we're in for some surprises: Sandy is listed, because of the following reasoning: (1) Sandy likes anyone/thing who likes cats, (2) cats like cats because everyone likes themselves, (3) therefore Sandy likes cats, and (4) therefore Sandy likes Sandy. Cats is an answer because of step (2), and finally, Sandy is an answer again, because of the clause about liking oneself. Notice that the result of the query is a list of solutions, where each solution corresponds to a different way of proving the query true. Sandy appears twice because there are two different ways of showing that Sandy likes Sandy. The order in which solutions appear is determined by the order of the search. Prolog searches for solutions in a top-down, left-to-right fashion. The clauses are searched from the top down, so the first clauses entered are the first ones tried. Within a clause, the body is searched left to right. In using the (likes Kim ?x) clause, Prolog would first try to find an x who likes Lee, and then see if x likes Kim.

The output from prove-all is not very pretty. We can fix that by defining a new function, top-level-prove, which calls prove-all as before, but then passes the list of solutions to show-prolog-solutions, which prints them in a more readable format. Note that show-prolog-solutions returns no values: (values). This means the read-eval-print loop will not print anything when (values) is the result of a top-level call.

```
(defmacro ?- (&rest goals)
  '(top-level-prove ',goals))

(defun top-level-prove (goals)
  "Prove the goals, and print variables readability."
  (show-prolog-solutions
   (variables-in goals)
   (prove-all goals no-bindings)))
```

```
(defun show-prolog-solutions (vars solutions)
  "Print the variables in each of the solutions."
  (if (null solutions)
      (format t "~&No.")
      (mapc #'(lambda (solution) (show-prolog-vars vars solution))
            solutions))
  (values))

(defun show-prolog-vars (vars bindings)
  "Print each variable with its binding."
  (if (null vars)
      (format t "~&Yes")
      (dolist (var vars)
        (format t "~&~a = ~a" var
                (subst-bindings bindings var))))
  (princ ";"))
```

Now let's try some queries:

```
> (?- (likes Sandy ?who))
?WHO = LEE;
?WHO = KIM;
?WHO = ROBIN;
?WHO = SANDY;
?WHO = CATS;
?WHO = SANDY;

> (?- (likes ?who Sandy))
?WHO = SANDY;
?WHO = KIM;
?WHO = SANDY;

> (?- (likes Robin Lee))
No.
```

The first query asks again whom Sandy likes, and the second asks who likes Sandy. The third asks for confirmation of a fact. The answer is “no,” because there are no clauses or facts that say Robin likes Lee. Here's another example, a list of pairs of people who are in a mutual liking relation. The last answer has an uninstantiated variable, indicating that everyone likes themselves.

```
> (?- (likes ?x ?y) (likes ?y ?x))
?Y = KIM
?X = SANDY;
?Y = SANDY
?X = SANDY;
?Y = SANDY
?X = SANDY;
?Y = SANDY
?X = KIM;
?Y = SANDY
?X = SANDY;
?Y = ?X3251
?X = ?X3251;
```

It makes sense in Prolog to ask open-ended queries like “what lists is 2 a member of?” or even “what items are elements of what lists?”

```
(?- (member 2 ?list))
(?- (member ?item ?list))
```

These queries are valid Prolog and will return solutions, but there will be an infinite number of them. Since our interpreter collects all the solutions into a single list before showing any of them, we will never get to see the solutions. The next section shows how to write a new interpreter that fixes this problem.



**Exercise 11.1[m]** The representation of relations has been a list whose first element is a symbol. However, for relations with no arguments, some people prefer to write ( $\leftarrow$  p q r) rather than ( $\leftarrow$  (p) (q) (r)). Make changes so that either form is acceptable.



**Exercise 11.2[m]** Some people find the  $\leftarrow$  notation difficult to read. Define macros rule and fact so that we can write:

```
(fact (likes Robin cats))
(rule (likes Sandy ?x) if (likes ?x cats))
```

## 11.3 Idea 3: Automatic Backtracking

The Prolog interpreter implemented in the last section solves problems by returning a list of all possible solutions. We'll call this a *batch* approach, because the answers are retrieved in one uninterrupted batch of processing. Sometimes that is just what you want, but other times a single solution will do. In real Prolog, solutions are presented one at a time, as they are found. After each solution is printed, the user has the option of asking for more solutions, or stopping. This is an *incremental* approach. The incremental approach will be faster when the desired solution is one of the first out of many alternatives. The incremental approach will even work when there is an infinite number of solutions. And if that is not enough, the incremental approach can be implemented so that it searches depth-first. This means that at any point it will require less storage space than the batch approach, which must keep all solutions in memory at once.

In this section we implement an incremental Prolog interpreter. One approach would be to modify the interpreter of the last section to use pipes rather than lists. With pipes, unnecessary computation is delayed, and even infinite lists can be expressed in a finite amount of time and space. We could change to pipes simply by changing the `mapcan` in `prove` and `prove-all` to `mappend-pipe` (page 286). The books by Winston and Horn (1988) and by Abelson and Sussman (1985) take this approach. We take a different one.

The first step is a version of `prove` and `prove-all` that return a single solution rather than a list of all possible solutions. This should be reminiscent of `achieve` and `achieve-all` from `gps` (chapter 4). Unlike `gps`, recursive subgoals and clobbered sibling goals are not checked for. However, `prove` is required to search systematically through all solutions, so it is passed an additional parameter: a list of other goals to achieve after achieving the first goal. This is equivalent to passing a continuation to `prove`. The result is that if `prove` ever succeeds, it means the entire top-level goal has succeeded. If it fails, it just means the program is backtracking and trying another sequence of choices. Note that `prove` relies on the fact that `fail` is `nil`, because of the way it uses `some`.

```
(defun prove-all (goals bindings)
  "Find a solution to the conjunction of goals."
  (cond ((eq bindings fail) fail)
        ((null goals) bindings)
        (t (prove (first goals) bindings (rest goals)))))

(defun prove (goal bindings other-goals)
  "Return a list of possible solutions to goal."
  (some #'(lambda (clause)
            (let ((new-clause (rename-variables clause)))
              (prove-all
                (append (clause-body new-clause) other-goals))))
```

---

```
(unify goal (clause-head new-clause) bindings)))
(get-clauses (predicate goal)))
```

If `prove` does succeed, it means a solution has been found. If we want more solutions, we need some way of making the process fail, so that it will backtrack and try again. One way to do that is to extend every query with a goal that will print out the variables, and ask the user if the computation should be continued. If the user says yes, then the goal *fails*, and backtracking starts. If the user says no, the goal succeeds, and since it is the final goal, the computation ends. This requires a brand new type of goal: one that is not matched against the data base, but rather causes some procedure to take action. In Prolog, such procedures are called *primitives*, because they are built-in to the language, and new ones may not be defined by the user. The user may, of course, define nonprimitive procedures that call upon the primitives.

In our implementation, primitives will be represented as Lisp functions. A predicate can be represented either as a list of clauses (as it has been so far) or as a single primitive. Here is a version of `prove` that calls primitives when appropriate:

```
(defun prove (goal bindings other-goals)
  "Return a list of possible solutions to goal."
  (let ((clauses (get-clauses (predicate goal))))
    (if (listp clauses)
        (some
         #'(lambda (clause)
             (let ((new-clause (rename-variables clause)))
               (prove-all
                 (append (clause-body new-clause) other-goals)
                 (unify goal (clause-head new-clause) bindings))))
         clauses)
        ;; The predicate's "clauses" can be an atom:
        ;; a primitive function to call
        (funcall clauses (rest goal) bindings
                  other-goals))))
```

Here is the version of `top-level-prove` that adds the primitive goal `show-prolog-vars` to the end of the list of goals. Note that this version need not call `show-prolog-solutions` itself, since the printing will be handled by the primitive for `show-prolog-vars`.

```
(defun top-level-prove (goals)
  (prove-all '(,@goals (show-prolog-vars ,@(variables-in goals)))
             no-bindings)
  (format t "~&No.")
  (values))
```

Here we define the primitive `show-prolog-vars`. All primitives must be functions of

three arguments: a list of arguments to the primitive relation (here a list of variables to show), a binding list for these arguments, and a list of pending goals. A primitive should either return `fail` or call `prove-all` to continue.

```
(defun show-prolog-vars (vars bindings other-goals)
  "Print each variable with its binding.
  Then ask the user if more solutions are desired."
  (if (null vars)
      (format t "~&Yes")
      (dolist (var vars)
        (format t "~&~a = ~a" var
                (subst-bindings bindings var))))
  (if (continue-p)
      fail
      (prove-all other-goals bindings)))
```

Since primitives are represented as entries on the `clauses` property of predicate symbols, we have to register `show-prolog-vars` as a primitive like this:

```
(setf (get 'show-prolog-vars 'clauses) 'show-prolog-vars)
```

Finally, the Lisp predicate `continue-p` asks the user if he or she wants to see more solutions:

```
(defun continue-p ()
  "Ask user if we should continue looking for solutions."
  (case (read-char)
    (#\; t)
    (#\. nil)
    (#\newline (continue-p))
    (otherwise
      (format t " Type ; to see more or . to stop")
      (continue-p))))
```

This version works just as well as the previous version on finite problems. The only difference is that the user, not the system, types the semicolons. The advantage is that we can now use the system on infinite problems as well. First, we'll ask what lists 2 is a member of:

```
> (?- (member 2 ?list))
?LIST = (2 . ?REST3302);
?LIST = (?X3303 2 . ?REST3307);
?LIST = (?X3303 ?X3308 2 . ?REST3312);
?LIST = (?X3303 ?X3308 ?X3313 2 . ?REST3317).
No.
```

The answers mean that 2 is a member of any list that starts with 2, or whose second element is 2, or whose third element is 2, and so on. The infinite computation was halted when the user typed a period rather than a semicolon. The “no” now means that there are no more answers to be printed; it will appear if there are no answers at all, if the user types a period, or if all the answers have been printed.

We can ask even more abstract queries. The answer to the next query says that an item is an element of a list when it is the first element, or the second, or the third, or the fourth, and so on.

```
> (?- (member ?item ?list))
?ITEM = ?ITEM3318
?LIST = (?ITEM3318 . ?REST3319);
?ITEM = ?ITEM3323
?LIST = (?X3320 ?ITEM3323 . ?REST3324);
?ITEM = ?ITEM3328
?LIST = (?X3320 ?X3325 ?ITEM3328 . ?REST3329);
?ITEM = ?ITEM3333
?LIST = (?X3320 ?X3325 ?X3330 ?ITEM3333 . ?REST3334).
No.
```

Now let's add the definition of the relation `length`:

```
(<- (length () 0))
(<- (length (?x . ?y) (1+ ?n)) (length ?y ?n))
```

Here are some queries showing that `length` can be used to find the second argument, the first, or both:

```
> (?- (length (a b c d) ?n))
?N = (1+ (1+ (1+ (1+ 0)))); 
No.

> (?- (length ?list (1+ (1+ 0))))
?LIST = (?X3869 ?X3872);
No.
```

```
> (?- (length ?list ?n))
?LIST = NIL
?N = 0;
?LIST = (?X3918)
?N = (1+ 0);
?LIST = (?X3918 ?X3921)
?N = (1+ (1+ 0)).
No.
```

The next two queries show the two lists of length two with a as a member. Both queries give the correct answer, a two-element list that either starts or ends with a. However, the behavior after generating these two solutions is quite different.

```
> (?- (length ?l (1+ (1+ 0))) (member a ?l))
?L = (A ?X4057);
?L = (?Y4061 A);
No.

> (?- (member a ?l) (length ?l (1+ (1+ 0))))
?L = (A ?X4081);
?L = (?Y4085 A);[Abort]
```

In the first query, `length` only generates one possible solution, the list with two unbound elements. `member` takes this solution and instantiates either the first or the second element to a.

In the second query, `member` keeps generating potential solutions. The first two partial solutions, where a is the first or second member of a list of unknown length, are extended by `length` to yield the solutions where the list has length two. After that, `member` keeps generating longer and longer lists, which `length` keeps rejecting. It is implicit in the definition of `member` that subsequent solutions will be longer, but because that is not explicitly known, they are all generated anyway and then explicitly tested and rejected by `length`.

This example reveals the limitations of Prolog as a pure logic-programming language. It turns out the user must be concerned not only about the logic of the problem but also with the flow of control. Prolog is smart enough to backtrack and find all solutions when the search space is small enough, but when it is infinite (or even very large), the programmer still has a responsibility to guide the flow of control. It is possible to devise languages that do much more in terms of automatic flow of control.<sup>4</sup> Prolog is a convenient and efficient middle ground between imperative languages and pure logic.

---

<sup>4</sup>See the MU-Prolog and NU-Prolog languages (Naish 1986).

### *Approaches to Backtracking*

Suppose you are asked to make a “small” change to an existing program. The problem is that some function,  $f$ , which was thought to be single-valued, is now known to return two or more valid answers in certain circumstances. In other words,  $f$  is nondeterministic. (Perhaps  $f$  is `sqrt`, and we now want to deal with negative numbers). What are your alternatives as a programmer? Five possibilities can be identified:

- Guess. Choose one possibility and discard the others. This requires a means of making the right guesses, or recovering from wrong guesses.
- Know. Sometimes you can provide additional information that is enough to decide what the right choice is. This means changing the calling function(s) to provide the additional information.
- Return a list. This means that the calling function(s) must be changed to expect a list of replies.
- Return a *pipe*, as defined in section 9.3. Again, the calling function(s) must be changed to expect a pipe.
- Guess and save. Choose one possibility and return it, but record enough information to allow computing the other possibilities later. This requires saving the current state of the computation as well as some information on the remaining possibilities.

The last alternative is the most desirable. It is efficient, because it doesn’t require computing answers that are never used. It is unobtrusive, because it doesn’t require changing the calling function (and the calling function’s calling function) to expect a list or pipe of answers. Unfortunately, it does have one major difficulty: there has to be a way of packaging up the current state of the computation and saving it away so that it can be returned to when the first choice does not work. For our Prolog interpreter, the current state is succinctly represented as a list of goals. In other problems, it is not so easy to summarize the entire state.

We will see in section 22.4 that the Scheme dialect of Lisp provides a function, `call-with-current-continuation`, that does exactly what we want: it packages the current state of the computation into a function, which can be stored away and invoked later. Unfortunately, there is no corresponding function in Common Lisp.

### *Anonymous Variables*

Before moving on, it is useful to introduce the notion of an *anonymous variable*. This is a variable that is distinct from all others in a clause or query, but which the

programmer does not want to bother to name. In real Prolog, the underscore is used for anonymous variables, but we will use a single question mark. The definition of `member` that follows uses anonymous variables for positions within terms that are not needed within a clause:

```
(<- (member ?item (?item . ?)))
(<- (member ?item (? . ?rest)) (member ?item ?rest))
```

However, we also want to allow several anonymous variables in a clause but still be able to keep each anonymous variable distinct from all other variables. One way to do that is to replace each anonymous variable with a unique variable. The function `replace-?-vars` uses `gensym` to do just that. It is installed in the top-level macros `<-` and `?-` so that all clauses and queries get the proper treatment.

```
(defmacro <- (&rest clause)
  "Add a clause to the data base."
  '(add-clause ',(replace-?-vars clause)))

(defmacro ?- (&rest goals)
  "Make a query and print answers."
  '(top-level-prove ',(replace-?-vars goals)))

(defun replace-?-vars (exp)
  "Replace any ? within exp with a var of the form ?123."
  (cond ((eq exp '?) (gensym "?"))
        ((atom exp) exp)
        (t (reuse-cons (replace-?-vars (first exp))
                      (replace-?-vars (rest exp))
                      exp))))
```

A named variable that is used only once in a clause can also be considered an anonymous variable. This is addressed in a different way in section 12.3.

## 11.4 The Zebra Puzzle

Here is an example of something Prolog is very good at: a logic puzzle. There are fifteen facts, or constraints, in the puzzle:

1. There are five houses in a line, each with an owner, a pet, a cigarette, a drink, and a color.
2. The Englishman lives in the red house.
3. The Spaniard owns the dog.

4. Coffee is drunk in the green house.
5. The Ukrainian drinks tea.
6. The green house is immediately to the right of the ivory house.
7. The Winston smoker owns snails.
8. Kools are smoked in the yellow house.
9. Milk is drunk in the middle house.
10. The Norwegian lives in the first house on the left.
11. The man who smokes Chesterfields lives next to the man with the fox.
12. Kools are smoked in the house next to the house with the horse.
13. The Lucky Strike smoker drinks orange juice.
14. The Japanese smokes Parliaments.
15. The Norwegian lives next to the blue house.

The questions to be answered are: who drinks water and who owns the zebra? To solve this puzzle, we first define the relations `nextto` (for “next to”) and `iright` (for “immediately to the right of”). They are closely related to `member`, which is repeated here.

```
(<- (member ?item (?item . ?rest)))
(<- (member ?item (?x . ?rest)) (member ?item ?rest))

(<- (nextto ?x ?y ?list) (iright ?x ?y ?list))
(<- (nextto ?x ?y ?list) (iright ?y ?x ?list))

(<- (iright ?left ?right (?left ?right . ?rest)))
(<- (iright ?left ?right (?x . ?rest))
     (iright ?left ?right ?rest))

(<- (= ?x ?x))
```

We also defined the identity relation, `=`. It has a single clause that says that any `x` is equal to itself. One might think that this implements `eq` or `equal`. Actually, since Prolog uses unification to see if the two arguments of a goal each unify with `?x`, this means that `=` is unification.

Now we are ready to define the zebra puzzle with a single (long) clause. The variable `?h` represents the list of five houses, and each house is represented by a term of the form (`house nationality pet cigarette drink color`). The variable `?w` is the water drinker, and `?z` is the zebra owner. Each of the 15 constraints in the puzzle is listed

in the body of zebra, although constraints 9 and 10 have been combined into the first one. Consider constraint 2, "The Englishman lives in the red house." This is interpreted as "there is a house whose nationality is Englishman and whose color is red, and which is a member of the list of houses": in other words, (member (house englishman ? ? ? red) ?h). The other constraints are similarly straightforward.

```

(<- (zebra ?h ?w ?z)
  ;; Each house is of the form:
  ;; (house nationality pet cigarette drink house-color)
  (= ?h ((house norwegian ? ? ? ?) ;1.10
        ?
        (house ? ? ? milk ?) ? ?)) ; 9
  (member (house englishman ? ? ? red) ?h) ; 2
  (member (house spaniard dog ? ? ?) ?h) ; 3
  (member (house ? ? ? coffee green) ?h) ; 4
  (member (house ukrainian ? ? tea ?) ?h) ; 5
  (right (house ? ? ? ? ivory) ; 6
         (house ? ? ? green) ?h)
  (member (house ? snails winston ? ?) ?h) ; 7
  (member (house ? ? kools ? yellow) ?h) ; 8
  (nextto (house ? ? chesterfield ? ?) ;11
           (house ? fox ? ? ?) ?h)
  (nextto (house ? ? kools ? ?) ;12
           (house ? horse ? ? ?) ?h)
  (member (house ? ? luckystrike orange-juice ?) ?h);13
  (member (house japanese ? parliaments ? ?) ?h) :14
  (nextto (house norwegian ? ? ? ?) ;15
           (house ? ? ? blue) ?h)
  ;; Now for the questions:
  (member (house ?w ? ? water ?) ?h) ;Q1
  (member (house ?z zebra ? ? ?) ?h)) ;Q2

```

Here's the query and solution to the puzzle:

```

> (?- (zebra ?houses ?water-drinker ?zebra-owner))
?HOUSES = ((HOUSE NORWEGIAN FOX KOOLS WATER YELLOW)
            (HOUSE UKRAINIAN HORSE CHESTERFIELD TEA BLUE)
            (HOUSE ENGLISHMAN SNAILS WINSTON MILK RED)
            (HOUSE SPANIARD DOG LUCKYSTRIKE ORANGE-JUICE IVORY)
            (HOUSE JAPANESE ZEBRA PARLIAMENTS COFFEE GREEN))
?WATER-DRINKER = NORWEGIAN
?ZEBRA-OWNER = JAPANESE.
No.

```

This took 278 seconds, and profiling (see page 288) reveals that the function prove was called 12,825 times. A call to prove has been termed a *logical inference*, so our system

is performing  $12825/278 = 46$  logical inferences per second, or LIPS. Good Prolog systems perform at 10,000 to 100,000 LIPS or more, so this is barely limping along.

Small changes to the problem can greatly affect the search time. For example, the relation `nextto` holds when the first house is immediately right of the second, or when the second is immediately right of the first. It is arbitrary in which order these clauses are listed, and one might think it would make no difference in which order they were listed. In fact, if we reverse the order of these two clauses, the execution time is roughly cut in half.

## 11.5 The Synergy of Backtracking and Unification

Prolog's backward chaining with backtracking is a powerful technique for generating the possible solutions to a problem. It makes it easy to implement a *generate-and-test* strategy, where possible solutions are considered one at a time, and when a candidate solution is rejected, the next is suggested. But generate-and-test is only feasible when the space of possible solutions is small.

In the zebra puzzle, there are five attributes for each of the five houses. Thus there are  $5!^5$ , or over 24 billion candidate solutions, far too many to test one at a time. It is the concept of unification (with the corresponding notion of a logic variable) that makes generate-and-test feasible on this puzzle. Instead of enumerating complete candidate solutions, unification allows us to specify *partial* candidates. We start out knowing that there are five houses, with the Norwegian living on the far left and the milk drinker in the middle. Rather than generating all complete candidates that satisfy these two constraints, we leave the remaining information vague, by unifying the remaining houses and attributes with anonymous logic variables. The next constraint (number 2) places the Englishman in the red house. Because of the way `member` is written, this first tries to place the Englishman in the leftmost house. This is rejected, because Englishman and Norwegian fail to unify, so the next possibility is considered, and the Englishman is placed in the second house. But no other features of the second house are specified—we didn't have to make separate guesses for the Englishman's house being green, yellow, and so forth. The search continues, filling in only as much as is necessary and backing up whenever a unification fails.

For this problem, unification serves the same purpose as the `delay` macro (page 281). It allows us to delay deciding the value of some attribute as long as possible, but to immediately reject a solution that tries to give two different values to the same attribute. That way, we save time if we end up backtracking before the computation is made, but we are still able to fill in the value later on.

It is possible to extend unification so that it is doing more work, and backtracking is doing less work. Consider the following computation:

```
(?- (length ?1 4)
     (member d ?1) (member a ?1) (member c ?1) (member b ?1)
     (= ?1 (a b c d)))
```

The first two lines generate permutations of the list (d a c b), and the third line tests for a permutation equal to (a b c d). Most of the work is done by backtracking. An alternative is to extend unification to deal with lists, as well as constants and variables. Predicates like `length` and `member` would be primitives that would have to know about the representation of lists. Then the first two lines of the above program would set `?1` to something like `#s(list :length 4 :members (d a c d))`. The third line would be a call to the extended unification procedure, which would further specify `?1` to be something like:

```
#s(list :length 4 :members (d a c d) :order (a b c d))
```

By making the unification procedure more complex, we eliminate the need for backtracking entirely.



**Exercise 11.3 [s]** Would a unification algorithm that delayed `member` tests be a good idea or a bad idea for the zebra puzzle?

## 11.6 Destructive Unification

As we saw in section 11.2, keeping track of a binding list of variables is a little tricky. It is also prone to inefficiency if the binding list grows large, because the list must be searched linearly, and because space must be allocated to hold the binding list. An alternative implementation is to change `unify` to a destructive operation. In this approach, there are no binding lists. Instead, each variable is represented as a structure that includes a field for its binding. When the variable is unified with another expression, the variable's binding field is modified to point to the expression. Such variables will be called `vars` to distinguish them from the implementation of variables as symbols starting with a question mark. `vars` are defined with the following code:

```
(defconstant unbound "Unbound")
(defstruct var name (binding unbound))
(defun bound-p (var) (not (eq (var-binding var) unbound)))
```

The macro `deref` gets at the binding of a variable, returning its argument when it is an

unbound variable or a nonvariable expression. It includes a loop because a variable can be bound to another variable, which in turn is bound to the ultimate value.

Normally, it would be considered bad practice to implement `deref` as a macro, since it could be implemented as an inline function, provided the caller was willing to write `(setf x (deref x))` instead of `(deref x)`. However, `deref` will appear in code generated by some versions of the Prolog compiler that will be presented in the next section. Therefore, to make the generated code look neater, I have allowed myself the luxury of the `deref` macro.

```
(defmacro deref (exp)
  "Follow pointers for bound variables."
  '(progn (loop while (and (var-p ,exp) (bound-p ,exp))
           do (setf ,exp (var-binding ,exp)))
          ,exp))
```

The function `unify!` below is the destructive version of `unify`. It is a predicate that returns true for success and false for failure, and has the side effect of altering variable bindings.

```
(defun unify! (x y)
  "Destructively unify two expressions"
  (cond ((eql (deref x) (deref y)) t)
        ((var-p x) (set-binding! x y))
        ((var-p y) (set-binding! y x))
        ((and (consp x) (consp y))
         (and (unify! (first x) (first y))
              (unify! (rest x) (rest y))))
        (t nil)))

(defun set-binding! (var value)
  "Set var's binding to value. Always succeeds (returns t)."
  (setf (var-binding var) value)
  t)
```

To make `vars` easier to read, we can install a `:print-function`:

```
(defstruct (var (:print-function print-var))
  name (binding unbound))

(defun print-var (var stream depth)
  (if (or (and (numberp *print-level*)
                (>= depth *print-level*))
           (var-p (deref var)))
      (format stream "?~a" (var-name var))
      (write var :stream stream)))
```

This is the first example of a carefully crafted :print-function. There are three things to notice about it. First, it explicitly writes to the stream passed as the argument. It does not write to a default stream. Second, it checks the variable depth against \*print-level\*, and prints just the variable name when the depth is exceeded. Third, it uses `write` to print the bindings. This is because `write` pays attention to the current values of \*print-escape\*, \*print-pretty\*, and so on. Other printing functions such as `prin1` or `print` do not pay attention to these variables.

Now, for backtracking purposes, we want to make `set-binding!` keep track of the bindings that were made, so they can be undone later:

```
(defvar *trail* (make-array 200 :fill-pointer 0 :adjustable t))

(defun set-binding! (var value)
  "Set var's binding to value, after saving the variable
  in the trail. Always returns t."
  (unless (eq var value)
    (vector-push-extend var *trail*)
    (setf (var-binding var) value))
  t)

(defun undo-bindings! (old-trail)
  "Undo all bindings back to a given point in the trail."
  (loop until (= (fill-pointer *trail*) old-trail)
    do (setf (var-binding (vector-pop *trail*)) unbound)))
```

Now we need a way of making new variables, where each one is distinct. That could be done by `gensym`-ing a new name for each variable, but a quicker solution is just to increment a counter. The constructor function ? is defined to generate a new variable with a name that is a new integer. This is not strictly necessary; we could have just used the automatically provided constructor `make-var`. However, I thought that the operation of providing new anonymous variable was different enough from providing a named variable that it deserved its own function. Besides, `make-var` may be less efficient, because it has to process the keyword arguments. The function ? has no arguments; it just assigns the default values specified in the slots of the `var` structure.

```
(defvar *var-counter* 0)

(defstruct (var (:constructor ? ())
                (:print-function print-var))
  (name (incf *var-counter*))
  (binding unbound))
```

A reasonable next step would be to use destructive unification to make a more efficient interpreter. This is left as an exercise, however, and instead we put the interpreter aside, and in the next chapter develop a compiler.

## 11.7 Prolog in Prolog

As stated at the start of this chapter, Prolog has many of the same features that make Lisp attractive for program development. Just as it is easy to write a Lisp interpreter in Lisp, it is easy to write a Prolog interpreter in Prolog. The following Prolog metainterpreter has three main relations. The relation `clause` is used to store clauses that make up the rules and facts that are to be interpreted. The relation `prove` is used to prove a goal. It calls `prove-all`, which attempts to prove a list of goals. `prove-all` succeeds in two ways: (1) if the list is empty, or (2) if there is some clause whose head matches the first goal, and if we can prove the body of that clause, followed by the remaining goals:

```
(<- (prove ?goal) (prove-all (?goal)))
(<- (prove-all nil))
(<- (prove-all (?goal . ?goals))
  (clause (<- ?goal . ?body))
  (concat ?body ?goals ?new-goals)
  (prove-all ?new-goals))
```

Now we add two clauses to the data base to define the `member` relation:

```
(<- (clause (<- (mem ?x (?x . ?y)))))
(<- (clause (<- (mem ?x (? . ?z)) (mem ?x ?z))))
```

Finally, we can prove a goal using our interpreter:

```
(?- (prove (mem ?x (1 2 3))))
?X = 1;
?X = 2;
?X = 3;
No.
```

## 11.8 Prolog Compared to Lisp

Many of the features that make Prolog a successful language for AI (and for program development in general) are the same as Lisp's features. Let's reconsider the list of features that make Lisp different from conventional languages (see page 25) and see what Prolog has to offer:

- *Built-in Support for Lists (and other data types).* New data types can be created easily using lists or structures (structures are preferred). Support for reading, printing, and accessing components is provided automatically. Numbers, symbols, and characters are also supported. However, because logic variables cannot be altered, certain data structures and operations are not provided. For example, there is no way to update an element of a vector in Prolog.
- *Automatic Storage Management.* The programmer can allocate new objects without worrying about reclaiming them. Reclaiming is usually faster in Prolog than in Lisp, because most data can be stack-allocated instead of heap-allocated.
- *Dynamic Typing.* Declarations are not required. Indeed, there is no standard way to make type declarations, although some implementations allow for them. Some Prolog systems provide only fixnums, so that eliminates the need for a large class of declarations.
- *First-Class Functions.* Prolog has no equivalent of `lambda`, but the built-in predicate `call` allows a term—a piece of data—to be called as a goal. Although backtracking choice points are not first-class objects, they can be used in a way very similar to continuations in Lisp.
- *Uniform Syntax.* Like Lisp, Prolog has a uniform syntax for both programs and data. This makes it easy to write interpreters and compilers in Prolog. While Lisp's prefix-operator list notation is more uniform, Prolog allows infix and postfix operators, which may be more natural for some applications.
- *Interactive Environment.* Expressions can be immediately evaluated. High-quality Prolog systems offer both a compiler and interpreter, along with a host of debugging tools.
- *Extensibility.* Prolog syntax is extensible. Because programs and data share the same format, it is possible to write the equivalent of macros in Prolog and to define embedded languages. However, it can be harder to ensure that the resulting code will be compiled efficiently. The details of Prolog compilation are implementation-dependent.

To put things in perspective, consider that Lisp is at once one of the highest-level languages available and a universal assembly language. It is a high-level language because it can easily capture data, functional, and control abstractions. It is a good assembly language because it is possible to write Lisp in a style that directly reflects the operations available on modern computers.

Prolog is generally not as efficient as an assembly language, but it can be more concise as a specification language, at least for some problems. The user writes specifications: lists of axioms that describe the relationships that can hold in the problem domain. If these specifications are in the right form, Prolog's automatic

backtracking can find a solution, even though the programmer does not provide an explicit algorithm. For other problems, the search space will be too large or infinite, or Prolog's simple depth-first search with backup will be too inflexible. In this case, Prolog must be used as a programming language rather than a specification language. The programmer must be aware of Prolog's search strategy, using it to implement an appropriate algorithm for the problem at hand.

Prolog, like Lisp, has suffered unfairly from some common myths. It has been thought to be an inefficient language because early implementations were interpreted, and because it has been used to write interpreters. But modern compiled Prolog can be quite efficient (see Warren et al. 1977 and Van Roy 1990). There is a temptation to see Prolog as a solution in itself rather than as a programming language. Those who take that view object that Prolog's depth-first search strategy and basis in predicate calculus is too inflexible. This objection is countered by Prolog programmers who use the facilities provided by the language to build more powerful search strategies and representations, just as one would do in Lisp or any other language.

## 11.9 History and References

Cordell Green (1968) was the first to articulate the view that mathematical results on theorem proving could be used to make deductions and thereby answer queries. However, the major technique in use at the time, resolution theorem proving (see Robinson 1965), did not adequately constrain search, and thus was not practical. The idea of goal-directed computing was developed in Carl Hewitt's work (1971) on the PLANNER language for robot problem solving. He suggested that the user provide explicit hints on how to control deduction.

At about the same time and independently, Alain Colmerauer was developing a system to perform natural language analysis. His approach was to weaken the logical language so that computationally complex statements (such as logical disjunctions) could not be made. Colmerauer and his group implemented the first Prolog interpreter using Algol-W in the summer of 1972 (see Roussel 1975). It was Roussel's wife, Jacqueline, who came up with the name Prolog as an abbreviation for "programmation en logique." The first large Prolog program was their natural language system, also completed that year (Colmerauer et al. 1973). For those who read English better than French, Colmerauer (1985) presents an overview of Prolog. Robert Kowalski is generally considered the coinventer of Prolog. His 1974 article outlines his approach, and his 1988 article is a historical review on the early logic programming work.

There are now dozens of text books on Prolog. In my mind, six of these stand out. Clocksin and Mellish's *Programming in Prolog* (1987) was the first and remains one of the best. Sterling and Shapiro's *The Art of Prolog* (1986) has more substantial examples but is not as complete as a reference. An excellent overview from a slightly

more mathematical perspective is Pereira and Shieber's *Prolog and Natural-Language Analysis* (1987). The book is worthwhile for its coverage of Prolog alone, and it also provides a good introduction to the use of logic programming for language understanding (see part V for more on this subject). O'Keefe's *The Craft of Prolog* (1990) shows a number of advanced techniques. O'Keefe is certainly one of the most influential voices in the Prolog community. He has definite views on what makes for good and bad coding style and is not shy about sharing his opinions. The reader is warned that this book evolved from a set of notes on the Clocksin and Mellish book, and the lack of organization shows in places. However, it contains advanced material that can be found nowhere else. Another collection of notes that has been organized into a book is Coelho and Cotta's *Prolog by Example*. Published in 1988, this is an update of their 1980 book, *How to Solve it in Prolog*. The earlier book was an underground classic in the field, serving to educate a generation of Prolog programmers. Both versions include a wealth of examples, unfortunately with little documentation and many typos. Finally, Ivan Bratko's *Prolog Programming for Artificial Intelligence* (1990) covers some introductory AI material from the Prolog perspective.

Maier and Warren's *Computing with Logic* (1988) is the best reference for those interested in implementing Prolog. It starts with a simple interpreter for a variable-free version of Prolog, and then moves up to the full language, adding improvements to the interpreter along the way. (Note that the second author, David S. Warren of Stonybrook, is different from David H. D. Warren, formerly at Edinburgh and now at Bristol. Both are experts on Prolog.)

Lloyd's *Foundations of Logic Programming* (1987) provides a theoretical explanation of the formal semantics of Prolog and related languages. Lassez et al. (1988) and Knight (1989) provide overviews of unification.

There have been many attempts to extend Prolog to be closer to the ideal of Logic Programming. The language MU-Prolog and NU-Prolog (Naish 1986) and Prolog III (Colmerauer 1990) are particularly interesting. The latter includes a systematic treatment of the  $\neq$  relation and an interpretation of infinite trees.

## 11.10 Exercises

-  **Exercise 11.4 [m]** It is somewhat confusing to see "no" printed after one or more valid answers have appeared. Modify the program to print "no" only when there are no answers at all, and "no more" in other cases.
-  **Exercise 11.5 [h]** At least six books (Abelson and Sussman 1985, Charniak and McDermott 1985, Charniak et al. 1986, Hennessey 1989, Wilensky 1986, and Winston and Horn 1988) present unification algorithms with a common error. They all have problems unifying  $(?x ?y a)$  with  $(?y ?x ?x)$ . Some of these texts assume that unify

will be called in a context where no variables are shared between the two arguments. However, they are still suspect to the bug, as the following example points out:

```
> (unify '(f (?x ?y a) (?y ?x ?x)) '(f ?z ?z))
((?Y . A) (?X . ?Y) (?Z ?X ?Y A))
```

Despite this subtle bug, I highly recommend each of the books to the reader. It is interesting to compare different implementations of the same algorithm. It turns out there are more similarities than differences. This indicates two things: (1) there is a generally agreed-upon style for writing these functions, and (2) good programmers sometimes take advantage of opportunities to look at other's code.

The question is: Can you give an informal proof of the correctness of the algorithm presented in this chapter? Start by making a clear statement of the specification. Apply that to the other algorithms, and show where they go wrong. Then see if you can prove that the `unify` function in this chapter is correct. Failing a complete proof, can you at least prove that the algorithm will always terminate? See Norvig 1991 for more on this problem.

-  **Exercise 11.6 [h]** Since logic variables are so basic to Prolog, we would like them to be efficient. In most implementations, structures are not the best choice for small objects. Note that variables only have two slots: the name and the binding. The binding is crucial, but the name is only needed for printing and is arbitrary for most variables. This suggests an alternative implementation. Each variable will be a cons cell of the variable's binding and an arbitrary marker to indicate the type. This marker would be checked by `variable-p`. Variable names can be stored in a hash table that is cleared before each query. Implement this representation for variables and compare it to the structure representation.
  
-  **Exercise 11.7 [m]** Consider the following alternative implementation for anonymous variables: Leave the macros `<-` and `?-` alone, so that anonymous variables are allowed in assertions and queries. Instead, change `unify` so that it lets anything match against an anonymous variable:

```
(defun unify (x y &optional (bindings no-bindings))
  "See if x and y match with given bindings."
  (cond ((eq bindings fail) fail)
        ((eql x y) bindings)
        ((or (eq x '?) (eq y '?)) bindings) ;***
        ((variable-p x) (unify-variable x y bindings))
        ((variable-p y) (unify-variable y x bindings))
        ((and (consp x) (consp y))
         (unify (rest x) (rest y)))
```

```
(unify (first x) (first y) bindings))
(t fail)))
```

Is this alternative correct? If so, give an informal proof. If not, give a counterexample.

 **Exercise 11.8 [h]** Write a version of the Prolog interpreter that uses destructive unification instead of binding lists.

 **Exercise 11.9 [m]** Write Prolog rules to express the terms father, mother, son, daughter, and grand- versions of each of them. Also define parent, child, wife, husband, brother, sister, uncle, and aunt. You will need to decide which relations are primitive (stored in the Prolog data base) and which are derived by rules.

For example, here's a definition of grandfather that says that G is the grandfather of C if G is the father of some P, who is the parent of C:

```
(<- (grandfather ?g ?c)
    (father ?g ?p)
    (parent ?p ?c))
```

 **Exercise 11.10 [m]** The following problem is presented in Wirth 1976:

*I married a widow (let's call her W) who has a grown-up daughter (call her D). My father (F), who visited us often, fell in love with my step-daughter and married her. Hence my father became my son-in-law and my step-daughter became my mother. Some months later, my wife gave birth to a son (S<sub>1</sub>), who became the brother-in-law of my father, as well as my uncle. The wife of my father, that is, my step-daughter, also had a son (S<sub>2</sub>).*

Represent this situation using the predicates defined in the previous exercise, verify its conclusions, and prove that the narrator of this tale is his own grandfather.

 **Exercise 11.11 [d]** Recall the example:

```
> (?- (length (a b c d) ?n))
?N = (1+ (1+ (1+ (1+ 0))));
```

It is possible to produce 4 instead of (1+ (1+ (1+ (1+ 0)))) by extending the notion of unification. Aït-Kaci et al. 1987 might give you some ideas how to do this.



**Exercise 11.12 [h]** The function `rename-variables` was necessary to avoid confusion between the variables in the first argument to `unify` and those in the second argument. An alternative is to change the `unify` so that it takes two binding lists, one for each argument, and keeps them separate. Implement this alternative.

## 11.11 Answers

**Answer 11.9** We will choose as primitives the unary predicates `male` and `female` and the binary predicates `child` and `married`. The former takes the child first; the latter takes the husband first. Given these primitives, we can make the following definitions:

```
(<- (father ?f ?c) (male ?f) (parent ?f ?c))
(<- (mother ?m ?c) (female ?m) (parent ?m ?c))
(<- (son ?s ?p) (male ?s) (parent ?p ?s))
(<- (daughter ?s ?p) (male ?s) (parent ?p ?s))

(<- (grandfather ?g ?c) (father ?g ?p) (parent ?p ?c))
(<- (grandmother ?g ?c) (mother ?g ?p) (parent ?p ?c))
(<- (grandson ?gs ?gp) (son ?gs ?p) (parent ?gp ?p))
(<- (granddaughter ?gd ?gp) (daughter ?gd ?p) (parent ?gp ?p))

(<- (parent ?p ?c) (child ?c ?p))
(<- (wife ?w ?h) (married ?h ?w))
(<- (husband ?h ?w) (married ?h ?w))

(<- (sibling ?x ?y) (parent ?p ?x) (parent ?p ?y))
(<- (brother ?b ?x) (male ?b) (sibling ?b ?x))
(<- (sister ?s ?x) (female ?s) (sibling ?s ?x))
(<- (uncle ?u ?n) (brother ?u ?p) (parent ?p ?n))
(<- (aunt ?a ?n) (sister ?a ?p) (parent ?p ?n))
```

Note that there is no way in Prolog to express a *true* definition. We would like to say that “P is the parent of C if and only if C is the child of P,” but Prolog makes us express the biconditional in one direction only.

**Answer 11.10** Because we haven't considered step-relations in the prior definitions, we have to extend the notion of parent to include step-parents. The definitions have to be written very carefully to avoid infinite loops. The strategy is to structure the defined terms into a strict hierarchy: the four primitives are at the bottom, then parent is defined in terms of the primitives, then the other terms are defined in terms of parent and the primitives.

We also provide a definition for son-in-law:

```
(<- (parent ?p ?c)      (married ?p ?w) (child ?c ?w))  
(<- (parent ?p ?c)      (married ?h ?p) (child ?c ?w))  
(<- (son-in-law ?s ?p) (parent ?p ?w) (married ?s ?w))
```

Now we add the information from the story. Note that we only use the four primitives male, female, married, and child:

```
(<- (male I))  (<- (male F))  (<- (male S1))  (<- (male S2))  
(<- (female W)) (<- (female D))  
(<- (married I W))  
(<- (married F D))  
(<- (child D W))  
(<- (child I F))  
(<- (child S1 I))  
(<- (child S2 F))
```

Now we are ready to make the queries:

```
> (?- (son-in-law F I))  
Yes.  
  
> (?- (mother D I))  
Yes.  
  
> (?- (uncle S1 I))  
Yes.  
  
> (?- (grandfather I I))  
Yes.
```

## *CHAPTER* 12

---

# Compiling Logic Programs

The end of chapter 11 introduced a new, more efficient representation for logic variables. It would be reasonable to build a new version of the Prolog interpreter incorporating this representation. However, chapter 9 has taught us that compilers run faster than interpreters and are not that much harder to build. Thus, this chapter will present a Prolog compiler that translates from Prolog to Lisp.

Each Prolog predicate will be translated into a Lisp function, and we will adopt the convention that a predicate called with a different number of arguments is a different predicate. If the symbol `p` can be called with either one or two arguments, we will need two Lisp functions to implement the two predicates. Following Prolog tradition, these will be called `p/1` and `p/2`.

The next step is to decide what the generated Lisp code should look like. It must unify the head of each clause against the arguments, and if the unification succeeds, it must call the predicates in the body. The difficult part is that the choice points have to be remembered. If a call to a predicate in the first clause fails, we must be able to return to the second clause and try again.

This can be done by passing in a *success continuation* as an extra argument to every predicate. This continuation represents the goals that remain unsolved, the other-goals argument of prove. For each clause in the predicate, if all the goals in a clause succeed, then we should call the success continuation. If a goal fails, we don't do anything special; we just go on to the next clause. There is one complication: after failing we have to undo any bindings made by unify!. Consider an example. The clauses

```
(<- (likes Robin cats))
(<- (likes Sandy ?x) (likes ?x cats))
(<- (likes Kim ?x) (likes ?x Lee) (likes ?x Kim))
```

could be compiled into this:

```
(defun likes/2 (?arg1 ?arg2 cont)
  ;;= First clause:
  (if (and (unify! ?arg1 'Robin) (unify! ?arg2 'cats))
      (funcall cont))
      (undo-bindings)
  ;;= Second clause:
  (if (unify! ?arg1 'Sandy)
      (likes/2 ?arg2 'cats cont))
      (undo-bindings)
  ;;= Third clause:
  (if (unify! ?arg1 'Kim)
      (likes/2 ?arg2 'Lee)
      #'(lambda () (likes/2 ?arg2 'Kim cont))))))
```

In the first clause, we just check the two arguments and, if the unifications succeed, call the continuation directly, because the first clause has no body. In the second clause, likes/2 is called recursively, to see if ?arg2 likes cats. If this succeeds, then the original goal succeeds, and the continuation cont is called. In the third clause, we have to call likes/2 recursively again, this time requesting that it check if ?arg2 likes Lee. If this check succeeds, then the continuation will be called. In this case, the continuation involves another call to likes/2, to check if ?arg2 likes Kim. If this succeeds, then the original continuation, cont, will finally be called.

Recall that in the Prolog interpreter, we had to append the list of pending goals, other-goals, to the goals in the body of the clause. In the compiler, there is no need to do an append. Instead, the continuation cont represents the other-goals, and the body of the clause is represented by explicit calls to functions.

Note that the code for `likes/2` given before has eliminated some unnecessary calls to `unify!`. The most obvious implementation would have one call to `unify!` for each argument. Thus, for the second clause, we would have the code:

```
(if (and (unify! ?arg1 'Sandy) (unify! ?arg2 ?x))
    (likes/2 ?x 'cats cont))
```

where we would need a suitable `let` binding for the variable `?x`.

## 12.1 A Prolog Compiler

This section presents the compiler summarized in figure 12.1. At the top level is the function `prolog-compile`, which takes a symbol, looks at the clauses defined for that symbol, and groups the clauses by arity. Each symbol/arity is compiled into a separate Lisp function by `compile-predicate`.

```
(defun prolog-compile (symbol &optional
                        (clauses (get-clauses symbol)))
  "Compile a symbol; make a separate function for each arity."
  (unless (null clauses)
    (let ((arity (relation-arity (clause-head (first clauses)))))
      ;; Compile the clauses with this arity
      (compile-predicate
        symbol arity (clauses-with-arity clauses #'= arity))
      ;; Compile all the clauses with any other arity
      (prolog-compile
        symbol (clauses-with-arity clauses #'/= arity))))
```

Three utility functions are included here:

```
(defun clauses-with-arity (clauses test arity)
  "Return all clauses whose head has given arity."
  (find-all arity clauses
    :key #'(lambda (clause)
              (relation-arity (clause-head clause)))
    :test test))
(defun relation-arity (relation)
  "The number of arguments to a relation.
  Example: (relation-arity '(p a b c)) => 3"
  (length (args relation)))
(defun args (x) "The arguments of a relation" (rest x))
```

The next step is to compile the clauses for a given predicate with a fixed arity into a

<b>Top-Level Functions</b>	
?-	Make a query, but compile everything first.
*trail*	<b>Special Variables</b> A list of all bindings made so far.
var	<b>Data Types</b> A box for a variable; can be destructively modified.
top-level-prove run-prolog prolog-compile-symbols prolog-compile compile-predicate compile-clause compile-body compile-call compile-arg compile-unify	<b>Major Functions</b> New version compiles everything first. Compile everything and call a Prolog function. Compile a list of Prolog symbols. Compile a symbol; make a separate function for each arity. Compile all the clauses for a given symbol/arity. Transform away the head and compile the resulting body. Compile the body of a clause. Compile a call to a Prolog predicate. Generate code for an argument to a goal in the body. Return code that tests if var and term unify.
clauses-with-arity relation-arity args make-parameters make-predicate make-= def-prolog-compiler-macro prolog-compiler-macro has-variable-p proper-listp maybe-add-undo-bindings bind-unbound-vars make-anonymous anonymous-variables-in compile-if compile-unify-variable bind-variables-in follow-binding bind-new-variables ignore	<b>Auxiliary Functions</b> Return all clauses whose head has given arity. The number of arguments to a relation. The arguments of a relation. Build a list of parameters. Build a symbol of the form name/arity. Build a unification relation. Define a compiler macro for Prolog. Fetch the compiler macro for a Prolog predicate. Is there a variable anywhere in the expression x? Is x a proper (non-dotted) list? Undo any bindings that need undoing. Add a let if needed. Replace variables that are only used once with ?. A list of anonymous variables. Compile an IF form. No else-part allowed. Compile the unification of a var. Bind all variables in exp to themselves. Get the ultimate binding of var according to bindings. Extend bindings to include any unbound variables. Do nothing—ignore the arguments.
unify! undo-bindings! binding-val symbol new-symbol find-anywhere	<b>Previously Defined Functions</b> Destructive unification (see section 11.6). Use the trail to backtrack, undoing bindings. Pick out the value part of a var/val binding. Create or find an interned symbol. Create a new uninterned symbol. Does item occur anywhere in tree?

Figure 12.1: Glossary for the Prolog Compiler

Lisp function. For now, that will be done by compiling each clause independently and wrapping them in a lambda with the right parameter list.

```
(defun compile-predicate (symbol arity clauses)
  "Compile all the clauses for a given symbol/arity
  into a single LISP function."
  (let ((predicate (make-predicate symbol arity))
        (parameters (make-parameters arity)))
    (compile
      (eval
        '(defun ,predicate (,@parameters cont)
          ,(mapcar #'(lambda (clause)
                      (compile-clause parameters clause 'cont))
                  clauses)))))

(defun make-parameters (arity)
  "Return the list (?arg1 ?arg2 ... ?arg-arity)"
  (loop for i from 1 to arity
        collect (new-symbol '?arg i)))

(defun make-predicate (symbol arity)
  "Return the symbol: symbol/arity"
  (symbol symbol '/ arity))
```

Now for the hard part: we must actually generate the code for a clause. Here again is an example of the code desired for one clause. We'll start by setting as a target the simple code:

```
(<- (likes Kim ?x) (likes ?x Lee) (likes ?x Kim))

(defun likes/2 (?arg1 ?arg2 cont)
  ...
  (if (and (unify! ?arg1 'Kim) (unify! ?arg2 ?x)
            (likes/2 ?arg2 'Lee
                     #'(lambda () (likes/2 ?x 'Kim))))
    ...)
```

but we'll also consider the possibility of upgrading to the improved code:

```
(defun likes/2 (?arg1 ?arg2 cont)
  ...
  (if (unify! ?arg1 'Kim)
    (likes/2 ?arg2 'Lee
             #'(lambda () (likes/2 ?arg2 'Kim))))
  ...)
```

One approach would be to write two functions, `compile-head` and `compile-body`,

and then combine them into the code (*if head body*). This approach could easily generate the prior code. However, let's allow ourselves to think ahead a little. If we eventually want to generate the improved code, we will need some communication between the head and the body. We will have to know that the head decided not to compile the unification of `?arg2` and `?x`, but because of this, the body will have to substitute `?arg2` for `?x`. That means that the `compile-head` function conceptually returns two values: the code for the head, and an indication of substitutions to perform in the body. This could be handled by explicitly manipulating multiple values, but it seems complicated.

An alternate approach is to eliminate `compile-head` and just write `compile-body`. This is possible if we in effect do a source-code transformation on the clause. Instead of treating the clause as:

```
(<- (likes Kim ?x)
     (likes ?x Lee) (likes ?x Kim))
```

we transform it to the equivalent:

```
(<- (likes ?arg1 ?arg2)
    (= ?arg1 Kim) (= ?arg2 ?x) (likes ?x Lee) (likes ?x Kim))
```

Now the arguments in the head of the clause match the arguments in the function `likes/2`, so there is no need to generate any code for the head. This makes things simpler by eliminating `compile-head`, and it is a better decomposition for another reason: instead of adding optimizations to `compile-head`, we will add them to the code in `compile-body` that handles `=`. That way, we can optimize calls that the user makes to `=`, in addition to the calls introduced by the source-code transformation.

To get an overview, the calling sequence of functions will turn out to be as follows:

```
prolog-compile
  compile-predicate
    compile-clause
      compile-body
        compile-call
        compile-arg
        compile-unify
        compile-arg
```

where each function calls the ones below it that are indented one level. We have already defined the first two functions. Here then is our first version of `compile-clause`:

```
(defun compile-clause (parms clause cont)
  "Transform away the head, and compile the resulting body."
  (compile-body
   (nconc
    (mapcar #'make-= parms (args (clause-head clause)))
    (clause-body clause))
   cont))

(defun make-= (x y) '(= ,x ,y))
```

The bulk of the work is in `compile-body`, which is a little more complicated. There are three cases. If there is no body, we just call the continuation. If the body starts with a call to `=`, we compile a call to `unify!`. Otherwise, we compile a call to a function, passing in the appropriate continuation.

However, it is worthwhile to think ahead at this point. If we want to treat `=` specially now, we will probably want to treat other goals specially later. So instead of explicitly checking for `=`, we will do a data-driven dispatch, looking for any predicate that has a `prolog-compiler-macro` property attached to it. Like Lisp compiler macros, the macro can decline to handle the goal. We will adopt the convention that returning `:pass` means the macro decided not to handle it, and thus it should be compiled as a normal goal.

```
(defun compile-body (body cont)
  "Compile the body of a clause."
  (if (null body)
      '(funcall ,cont)
      (let* ((goal (first body))
             (macro (prolog-compiler-macro (predicate goal)))
             (macro-val (if macro
                            (funcall macro goal (rest body) cont))))
        (if (and macro (not (eq macro-val :pass)))
            macro-val
            (compile-call
             (make-predicate (predicate goal)
                            (relation-arity goal))
             (mapcar #'(lambda (arg) (compile-arg arg))
                     (args goal)))
             (if (null (rest body))
                 cont
                 #'(lambda ()
                     ,(compile-body (rest body) cont)))))))
  (defun compile-call (predicate args cont)
    "Compile a call to a prolog predicate."
    '(,predicate ,@args ,cont))
```

```
(defun prolog-compiler-macro (name)
  "Fetch the compiler macro for a Prolog predicate."
  ;;= Note NAME is the raw name, not the name/arity
  (get name 'prolog-compiler-macro))

(defmacro def-prolog-compiler-macro (name arglist &body body)
  "Define a compiler macro for Prolog."
  '(setf (get ',name 'prolog-compiler-macro)
        #'(lambda ,arglist ,body)))

(def-prolog-compiler-macro = (goal body cont)
  (let ((args (args goal)))
    (if (/= (length args) 2)
        :pass
        '(if ,(compile-unify (first args) (second args))
            ,(compile-body body cont)))))

(defun compile-unify (x y)
  "Return code that tests if var and term unify."
  '(unify! ,(compile-arg x) ,(compile-arg y)))
```

All that remains is `compile-arg`, a function to compile the arguments to goals in the body. There are three cases to consider, as shown in the compilation to the argument of `q` below:

1 ( <code>&lt;- (p ?x) (q ?x)</code> )	<code>(q/1 ?x cont)</code>
2 ( <code>&lt;- (p ?x) (q (f a b))</code> )	<code>(q/1 '(f a b) cont)</code>
3 ( <code>&lt;- (p ?x) (q (f ?x b))</code> )	<code>(q/1 (list 'f ?x 'b) cont)</code>

In case 1, the argument is a variable, and it is compiled as is. In case 2, the argument is a constant expression (one without any variables) that compiles into a quoted expression. In case 3, the argument contains a variable, so we have to generate code that builds up the expression. Case 3 is actually split into two in the list below: one compiles into a call to `list`, and the other a call to `cons`. It is important to remember that the goal `(q (f ?x b))` does *not* involve a call to the function `f`. Rather, it involves the term `(f ?x b)`, which is just a list of three elements.

```
(defun compile-arg (arg)
  "Generate code for an argument to a goal in the body."
  (cond ((variable-p arg) arg)
        ((not (has-variable-p arg)) ',arg)
        ((proper-listp arg)
         '(list ,(mapcar #'compile-arg arg)))
        (t '(cons ,(compile-arg (first arg))
                  ,(compile-arg (rest arg))))))
```

```
(defun has-variable-p (x)
  "Is there a variable anywhere in the expression x?"
  (find-if-anywhere #'variable-p x))

(defun proper-listp (x)
  "Is x a proper (non-dotted) list?"
  (or (null x)
      (and (consp x) (proper-listp (rest x)))))
```

Let's see how it works. We will consider the following clauses:

```
(<- (likes Robin cats))
(<- (likes Sandy ?x) (likes ?x cats))
(<- (likes Kim ?x) (likes ?x Lee) (likes ?x Kim))

(<- (member ?item (?item . ?rest)))
(<- (member ?item (?x . ?rest)) (member ?item ?rest))
```

Here's what prolog-compile gives us:

```
(DEFUN LIKES/2 (?ARG1 ?ARG2 CONT)
  (IF (UNIFY! ?ARG1 'ROBIN)
      (IF (UNIFY! ?ARG2 'CATS)
          (FUNCALL CONT)))
  (IF (UNIFY! ?ARG1 'SANDY)
      (IF (UNIFY! ?ARG2 ?X)
          (LIKES/2 ?X 'CATS CONT)))
  (IF (UNIFY! ?ARG1 'KIM)
      (IF (UNIFY! ?ARG2 ?X)
          (LIKES/2 ?X 'LEE (LAMBDA ()
            (LIKES/2 ?X 'KIM CONT))))))

(DEFUN MEMBER/2 (?ARG1 ?ARG2 CONT)
  (IF (UNIFY! ?ARG1 ?ITEM)
      (IF (UNIFY! ?ARG2 (CONS ?ITEM ?REST))
          (FUNCALL CONT)))
  (IF (UNIFY! ?ARG1 ?ITEM)
      (IF (UNIFY! ?ARG2 (CONS ?X ?REST))
          (MEMBER/2 ?ITEM ?REST CONT))))
```

## 12.2 Fixing the Errors in the Compiler

There are some problems in this version of the compiler:

- We forgot to undo the bindings after each call to `unify!`.
- The definition of `undo-bindings!` defined previously requires as an argument an index into the `*trail*` array. So we will have to save the current top of the trail when we enter each function.
- Local variables, such as `?x`, were used without being introduced. They should be bound to new variables.

Undoing the bindings is simple: we add a single line to `compile-predicate`, a call to the function `maybe-add-undo-bindings`. This function inserts a call to `undo-bindings!` after every failure. If there is only one clause, no undoing is necessary, because the predicate higher up in the calling sequence will do it when it fails. If there are multiple clauses, the function wraps the whole function body in a `let` that captures the initial value of the trail's fill pointer, so that the bindings can be undone to the right point. Similarly, we can handle the unbound-variable problem by wrapping a call to `bind-unbound-vars` around each compiled clause:

```
(defun compile-predicate (symbol arity clauses)
  "Compile all the clauses for a given symbol/arity
  into a single LISP function."
  (let ((predicate (make-predicate symbol arity))
        (parameters (make-parameters arity)))
    (compile
      (eval
        '(defun ,predicate (,@parameters cont)
          ..,(maybe-add-undo-bindings                      ;***
            (mapcar #'(lambda (clause)
                        (compile-clause parameters
                                          clause 'cont))
                    clauses)))))))

(defun compile-clause (parms clause cont)
  "Transform away the head, and compile the resulting body."
  (bind-unbound-vars                                ;***
    parms                                         ;***
    (compile-body
      (nconc
        (mapcar #'make-= parms (args (clause-head clause)))
        (clause-body clause))
      cont)))
```

```
(defun maybe-add-undo-bindings (compiled-exp)
  "Undo any bindings that need undoing.
  If there are any, bind the trail before we start."
  (if (length=1 compiled-exp)
      compiled-exp
      '(((let ((old-trail (fill-pointer *trail*)))
          ,(first compiled-exp)
          ,@(loop for exp in (rest compiled-exp)
                  collect '(undo-bindings! old-trail)
                  collect exp)))))

(defun bind-unbound-vars (parameters exp)
  "If there are any variables in exp (besides the parameters)
  then bind them to new vars."
  (let ((exp-vars (set-difference (variables-in exp)
                                   parameters)))
    (if exp-vars
        '(let ,(mapcar #'(lambda (var) '(,var (?)))
                      exp-vars)
           ,exp)
        exp)))
```

With these improvements, here's the code we get for `likes` and `member`:

```
(DEFUN LIKES/2 (?ARG1 ?ARG2 CONT)
  (LET ((OLD-TRAIL (FILL-POINTER *TRAIL*)))
    (IF (UNIFY! ?ARG1 'ROBIN)
        (IF (UNIFY! ?ARG2 'CATS)
            (FUNCALL CONT)))
    (UNDO-BINDINGS! OLD-TRAIL)
    (LET ((?X (?)))
      (IF (UNIFY! ?ARG1 'SANDY)
          (IF (UNIFY! ?ARG2 ?X)
              (LIKES/2 ?X 'CATS CONT))))
    (UNDO-BINDINGS! OLD-TRAIL)
    (LET ((?X (?)))
      (IF (UNIFY! ?ARG1 'KIM)
          (IF (UNIFY! ?ARG2 ?X)
              (LIKES/2 ?X 'LEE (LAMBDA ()
                (LIKES/2 ?X 'KIM CONT)))))))
```

```
(DEFUN MEMBER/2 (?ARG1 ?ARG2 CONT)
  (LET ((OLD-TRAIL (FILL-POINTER *TRAIL*)))
    (LET ((?ITEM (?))
          (?REST (?)))
        (IF (UNIFY! ?ARG1 ?ITEM)
            (IF (UNIFY! ?ARG2 (CONS ?ITEM ?REST))
                (FUNCALL CONT))))
      (UNDO-BINDINGS! OLD-TRAIL)
      (LET ((?X (?))
            (?ITEM (?))
            (?REST (?)))
        (IF (UNIFY! ?ARG1 ?ITEM)
            (IF (UNIFY! ?ARG2 (CONS ?X ?REST))
                (MEMBER/2 ?ITEM ?REST CONT)))))))
```

## 12.3 Improving the Compiler

This is fairly good, although there is still room for improvement. One minor improvement is to eliminate unneeded variables. For example, `?rest` in the first clause of `member` and `?x` in the second clause are bound to new variables—the result of the `(?)` call—and then only used once. The generated code could be made a little tighter by just putting `(?)` inline, rather than binding it to a variable and then referencing that variable. There are two parts to this change: updating `compile-arg` to compile an anonymous variable inline, and changing the `<-` macro so that it converts all variables that only appear once in a clause into anonymous variables:

```
(defmacro <- (&rest clause)
  "Add a clause to the data base."
  '(add-clause ',(make-anonymous clause)))
(defun compile-arg (arg)
  "Generate code for an argument to a goal in the body."
  (cond ((eq arg '?) '(?)) ;***
        ((variable-p arg) arg)
        ((not (has-variable-p arg)) ',arg)
        ((proper-listp arg)
         `',(list .,(mapcar #'compile-arg arg)))
        (t `(cons ,(compile-arg (first arg))
                  ,(compile-arg (rest arg))))))
(defun make-anonymous (exp &optional
                        (anon-vars (anonymous-variables-in exp)))
  "Replace variables that are only used once with ?."
  (cond ((consp exp)
         (reuse-cons (make-anonymous (first exp) anon-vars)
```

```

        (make-anonymous (rest exp) anon-vars)
        exp))
((member exp anon-vars) '?)
(t exp)))

```

Finding anonymous variables is tricky. The following function keeps two lists: the variables that have been seen once, and the variables that have been seen twice or more. The local function `walk` is then used to walk over the tree, recursively considering the components of each cons cell and updating the two lists as each variable is encountered. This use of local functions should be remembered, as well as an alternative discussed in exercise 12.23 on page 428.

```

(defun anonymous-variables-in (tree)
  "Return a list of all variables that occur only once in tree."
  (let ((seen-once nil)
        (seen-more nil))
    (labels ((walk (x)
                  (cond
                    ((variable-p x)
                     (cond ((member x seen-once)
                            (setf seen-once (delete x seen-once))
                            (push x seen-more))
                           ((member x seen-more) nil)
                           (t (push x seen-once))))
                    ((consp x)
                     (walk (first x))
                     (walk (rest x))))))
      (walk tree)
      seen-once)))

```

Now `member` compiles into this:

```

(DEFUN MEMBER/2 (?ARG1 ?ARG2 CONT)
  (LET ((OLD-TRAIL (FILL-POINTER *TRAIL*)))
    (LET ((?ITEM (?)))
      (IF (UNIFY! ?ARG1 ?ITEM)
          (IF (UNIFY! ?ARG2 (CONS ?ITEM (?)))
              (FUNCALL CONT)))
        (UNDO-BINDINGS! OLD-TRAIL)
        (LET ((?ITEM (?))
              (?REST (?)))
          (IF (UNIFY! ?ARG1 ?ITEM)
              (IF (UNIFY! ?ARG2 (CONS (?) ?REST))
                  (MEMBER/2 ?ITEM ?REST CONT)))))))

```

## 12.4 Improving the Compilation of Unification

Now we turn to the improvement of `compile-unify`. Recall that we want to eliminate certain calls to `unify!` so that, for example, the first clause of `member`:

```
(<- (member ?item (?item . ?rest)))
```

compiles into:

```
(LET ((?ITEM (?)))
  (IF (UNIFY! ?ARG1 ?ITEM)
    (IF (UNIFY! ?ARG2 (CONS ?ITEM (?)))
      (FUNCALL CONT))))
```

when it could compile to the more efficient:

```
(IF (UNIFY! ?ARG2 (CONS ?ARG1 (?)))
  (FUNCALL CONT))
```

Eliminating the unification in one goal has repercussions in other goals later on, so we will need to keep track of expressions that have been unified together. We have a design choice. Either `compile-unify` can modify a global state variable, or it can return multiple values. On the grounds that global variables are messy, we make the second choice: `compile-unify` will take a binding list as an extra argument and will return two values, the actual code and an updated binding list. We will expect that other related functions will have to be modified to deal with these multiple values.

When `compile-unify` is first called in our example clause, it is asked to unify `?arg1` and `?item`. We want it to return no code (or more precisely, the trivially true test, `t`). For the second value, it should return a new binding list, with `?item` bound to `?arg1`. That binding will be used to replace `?item` with `?arg1` in subsequent code.

How do we know to bind `?item` to `?arg1` rather than the other way around? Because `?arg1` is already bound to something—the value passed in to `member`. We don't know what this value is, but we can't ignore it. Thus, the initial binding list will have to indicate that the parameters are bound to something. A simple convention is to bind the parameters to themselves. Thus, the initial binding list will be:

```
((?arg1 . ?arg1) (?arg2 . ?arg2))
```

We saw in the previous chapter (page 354) that binding a variable to itself can lead to problems; we will have to be careful.

Besides eliminating unifications of new variables against parameters, there are quite a few other improvements that can be made. For example, unifications involv-

ing only constants can be done at compile time. The call  $(= (f a) (f a))$  always succeeds, while  $(= 3 4)$  always fails. In addition, unification of two cons cells can be broken into components at compile time:  $(= (f ?x) (f a))$  reduces to  $(= ?x a)$  and  $(= f f)$ , where the latter trivially succeeds. We can even do some occurs checking at compile time:  $(= ?x (f ?x))$  should fail.

The following table lists these improvements, along with a breakdown for the cases of unifying a bound ( $?arg1$ ) or unbound ( $?x$ ) variable against another expression. The first column is the unification call, the second is the generated code, and the third is the bindings that will be added as a result of the call:

Unification	Code	Bindings
1 $(= 3 3)$	t	—
2 $(= 3 4)$	nil	—
3 $(= (f ?x) (?p 3))$	t	$(?x . 3) (?p . f)$
4 $(= ?arg1 ?y)$	t	$(?y . ?arg1)$
5 $(= ?arg1 ?arg2)$	$(unify! ?arg1 ?arg2)$	$(?arg1 . ?arg2)$
6 $(= ?arg1 3)$	$(unify! ?arg1 3)$	$(?arg1 . 3)$
7 $(= ?arg1 (f ?y))$	$(unify! ?arg1 ...)$	$(?y . ?y)$
8 $(= ?x ?y)$	t	$(?x . ?y)$
9 $(= ?x 3)$	t	$(?x . 3)$
10 $(= ?x (f ?y))$	$(unify! ?x ...)$	$(?y . ?y)$
11 $(= ?x (f ?x))$	nil	—
12 $(= ?x ?)$	t	—

From this table we can craft our new version of `compile-unify`. The first part is fairly easy. It takes care of the first three cases in this table and makes sure that `compile-unify-variable` is called with a variable as the first argument for the other cases.

```
(defun compile-unify (x y bindings)
  "Return 2 values: code to test if x and y unify,
  and a new binding list."
  (cond
    ;; Unify constants and conses: ; Case
    ((not (or (has-variable-p x) (has-variable-p y))) ; 1,2
     (values (equal x y) bindings))
    ((and (consp x) (consp y)) ; 3
     (multiple-value-bind (code1 bindings1)
       (compile-unify (first x) (first y) bindings)
       (multiple-value-bind (code2 bindings2)
         (compile-unify (rest x) (rest y) bindings1)
         (values (compile-if code1 code2) bindings2))))
    ;; Here x or y is a variable. Pick the right one:
    ((variable-p x) (compile-unify-variable x y bindings))
    (t (compile-unify-variable y x bindings))))
```

```
(defun compile-if (pred then-part)
  "Compile a Lisp IF form. No else-part allowed."
  (case pred
    ((t) then-part)
    ((nil) nil)
    (otherwise `(if ,pred ,then-part))))
```

The function `compile-unify-variable` following is one of the most complex we have seen. For each argument, we see if it has a binding (the local variables `xb` and `yb`), and then use the bindings to get the value of each argument (`x1` and `y1`). Note that for either an unbound variable or one bound to itself, `x` will equal `x1` (and the same for `y` and `y1`). If either of the pairs of values is not equal, we should use the new ones (`x1` or `y1`), and the clause commented `deref` does that. After that point, we just go through the cases, one at a time. It turns out that it was easier to change the order slightly from the preceding table, but each clause is commented with the corresponding number:

```
(defun compile-unify-variable (x y bindings)
  "X is a variable, and Y may be."
  (let* ((xb (follow-binding x bindings))
         (x1 (if xb (cdr xb) x))
         (yb (if (variable-p y) (follow-binding y bindings)))
         (y1 (if yb (cdr yb) y)))
    (cond
      ((or (eq x '?) (eq y '?)) (values t bindings)) ; Case: 12
      ((not (and (equal x x1) (equal y y1))) ; deref
       (compile-unify x1 y1 bindings))
      ((find-anywhere x1 y1) (values nil bindings)) ; 11
      ((consp y1) ; 7,10
       (values '(unify! ,x1 ,(compile-arg y1 bindings))
              (bind-variables-in y1 bindings)))
      ((not (null xb))
       ;; i.e. x is an ?arg variable
       (if (and (variable-p y1) (null yb))
           (values 't (extend-bindings y1 x1 bindings)) ; 4
           (values '(unify! ,x1 ,(compile-arg y1 bindings))
                  (extend-bindings x1 y1 bindings)))) ; 5,6
      ((not (null yb))
       (compile-unify-variable y1 x1 bindings))
      (t (values 't (extend-bindings x1 y1 bindings)))))) ; 8,9
```

Take some time to understand just how this function works. Then go on to the following auxiliary functions:

```
(defun bind-variables-in (exp bindings)
  "Bind all variables in exp to themselves, and add that to
  bindings (except for variables already bound)."
  (dolist (var (variables-in exp))
    (unless (get-binding var bindings)
      (setf bindings (extend-bindings var var bindings))))
  bindings)

(defun follow-binding (var bindings)
  "Get the ultimate binding of var according to bindings."
  (let ((b (get-binding var bindings)))
    (if (eq (car b) (cdr b))
        b
        (or (follow-binding (cdr b) bindings)
            b))))
```

Now we need to integrate the new `compile-unify` into the rest of the compiler. The problem is that the new version takes an extra argument and returns an extra value, so all the functions that call it need to be changed. Let's look again at the calling sequence:

```
prolog-compile
  compile-predicate
  compile-clause
  compile-body
  compile-call
  compile-arg
  compile-unify
  compile-arg
```

First, going downward, we see that `compile-arg` needs to take a binding list as an argument, so that it can look up and substitute in the appropriate values. But it will not alter the binding list, so it still returns one value:

```
(defun compile-arg (arg bindings)
  "Generate code for an argument to a goal in the body."
  (cond ((eq arg '?) '(?))
        ((variable-p arg)
         (let ((binding (get-binding arg bindings)))
           (if (and (not (null binding))
                     (not (eq arg (binding-val binding))))
               (compile-arg (binding-val binding) bindings)
               arg)))
        ((not (find-if-anywhere #'variable-p arg)) ',arg)
        ((proper-listp arg)
         '(list .,(mapcar #'(lambda (a) (compile-arg a bindings))
```

```
arg)))
(t '(cons ,(compile-arg (first arg) bindings)
,(compile-arg (rest arg) bindings))))
```

Now, going upward, `compile-body` needs to take a binding list and pass it on to various functions:

```
(defun compile-body (body cont bindings)
  "Compile the body of a clause."
  (cond
    ((null body)
     '(funcall ,cont))
    (t (let* ((goal (first body))
              (macro (prolog-compiler-macro (predicate goal)))
              (macro-val (if macro
                             (funcall macro goal (rest body)
                                      cont-bindings))))
        (if (and macro (not (eq macro-val :pass)))
            macro-val
            (compile-call
              (make-predicate (predicate goal)
                            (relation-arity goal))
              (mapcar #'(lambda (arg)
                          (compile-arg arg bindings))
                      (args goal)))
            (if (null (rest body))
                cont
                '#'(lambda ()
                      ,(compile-body
                        (rest body) cont
                        (bind-new-variables bindings goal)))))))))))
```

The function `bind-new-variables` takes any variables mentioned in the goal that have not been bound yet and binds these variables to themselves. This is because the goal, whatever it is, may bind its arguments.

```
(defun bind-new-variables (bindings goal)
  "Extend bindings to include any unbound variables in goal."
  (let ((variables (remove-if #'(lambda (v) (assoc v bindings))
                                (variables-in goal))))
    (nconc (mapcar #'self-cons variables) bindings)))
(defun self-cons (x) (cons x x))
```

One of the functions that needs to be changed to accept a binding list is the compiler macro for `=`:

```
(def-prolog-compiler-macro = (goal body cont bindings)
  "Compile a goal which is a call to =."
  (let ((args (args goal)))
    (if (/= (length args) 2)
        :pass ;; decline to handle this goal
        (multiple-value-bind (code1 bindings1)
            (compile-unify (first args) (second args) bindings)
            (compile-if
              code1
              (compile-body body cont bindings1))))))

```

The last step upward is to change `compile-clause` so that it starts everything off by passing into `compile-body` a binding list with all the parameters bound to themselves:

```
(defun compile-clause (parms clause cont)
  "Transform away the head, and compile the resulting body."
  (bind-unbound-vars
    parms
    (compile-body
      (nconc
        (mapcar #'make-= parms (args (clause-head clause)))
        (clause-body clause))
      cont
      (mapcar #'self-cons parms))))           ;***
```

Finally, we can see the fruits of our efforts:

```
(DEFUN MEMBER/2 (?ARG1 ?ARG2 CONT)
  (LET ((OLD-TRAIL (FILL-POINTER *TRAIL*)))
    (IF (UNIFY! ?ARG2 (CONS ?ARG1 (?)))
        (FUNCALL CONT))
    (UNDO-BINDINGS! OLD-TRAIL)
    (LET ((?REST (?)))
      (IF (UNIFY! ?ARG2 (CONS (?) ?REST))
          (MEMBER/2 ?ARG1 ?REST CONT)))))

(DEFUN LIKES/2 (?ARG1 ?ARG2 CONT)
  (LET ((OLD-TRAIL (FILL-POINTER *TRAIL*)))
    (IF (UNIFY! ?ARG1 'ROBIN)
        (IF (UNIFY! ?ARG2 'CATS)
            (FUNCALL CONT))
        (UNDO-BINDINGS! OLD-TRAIL)
        (IF (UNIFY! ?ARG1 'SANDY)
            (LIKES/2 ?ARG2 'CATS CONT))
        (UNDO-BINDINGS! OLD-TRAIL)
        (IF (UNIFY! ?ARG1 'KIM)
            (LIKES/2 ?ARG2 'LEE (LAMBDA ()
              (LIKES/2 ?ARG2 'KIM CONT)))))))
```

## 12.5 Further Improvements to Unification

Could `compile-unify!` be improved yet again? If we insist that it call `unify!`, it seems that it can't be made much better. However, we could improve it by in effect compiling `unify!`. This is a key idea in the Warren Abstract Machine, or WAM, which is the most commonly used model for Prolog compilers.

We call `unify!` in four cases (5, 6, 7, and 10), and in each case the first argument is a variable, and we know something about the second argument. But the first thing `unify!` does is redundantly test if the first argument is a variable. We could eliminate unnecessary tests by calling more specialized functions rather than the general-purpose function `unify!`. Consider this call:

```
(unify! ?arg2 (cons ?arg1 (?)))
```

If `?arg2` is an unbound variable, this code is appropriate. But if `?arg2` is a constant atom, we should fail immediately, without allowing `cons` and `?` to generate garbage. We could change the test to:

```
(and (consp-or-variable-p ?arg2)
      (unify-first! ?arg2 ?arg1)
      (unify-rest! ?arg2 (?)))
```

with suitable definitions for the functions referenced here. This change should speed execution time and limit the amount of garbage generated. Of course, it makes the generated code longer, so that could slow things down if the program ends up spending too much time bringing the code to the processor.

 **Exercise 12.1 [h]** Write definitions for `consp-or-variable-p`, `unify-first!`, and `unify-rest!`, and change the compiler to generate code like that outlined previously. You might want to look at the function `compile-rule` in section 9.6, starting on page 300. This function compiled a call to `pat-match` into individual tests; now we want to do the same thing to `unify!`. Run some benchmarks to compare the altered compiler to the original version.

 **Exercise 12.2 [h]** We can gain some more efficiency by keeping track of which variables have been dereferenced and calling an appropriate unification function: either one that dereferences the argument or one that assumes the argument has already been dereferenced. Implement this approach.

 **Exercise 12.3 [m]** What code is generated for `(= (f (g ?x) ?y) (f ?y (?p a)))`?

What more efficient code represents the same unification? How easy is it to change the compiler to get this more efficient result?

- ?
- Exercise 12.4 [h]** In retrospect, it seems that binding variables to themselves, as in `(?arg1 . ?arg1)`, was not such a good idea. It complicates the meaning of bindings, and prohibits us from using existing tools. For example, I had to use `find-anywhere` instead of `occur-check` for case 11, because `occur-check` expects a noncircular binding list. But `find-anywhere` does not do as complete a job as `occur-check`. Write a version of `compile-unify` that returns three values: the code, a noncircular binding list, and a list of variables that are bound to unknown values.
  
- ?
- Exercise 12.5 [h]** An alternative to the previous exercise is not to use binding lists at all. Instead, we could pass in a list of equivalence classes—that is, a list of lists, where each sublist contains one or more elements that have been unified. In this approach, the initial equivalence class list would be `((?arg1) (?arg2))`. After unifying `?arg1` with `?x`, `?arg2` with `?y`, and `?x` with 4, the list would be `((4 ?arg1 ?x) (?arg2 ?y))`. This assumes the convention that the canonical member of an equivalence class (the one that will be substituted for all others) comes first. Implement this approach. What advantages and disadvantages does it have?

## 12.6 The User Interface to the Compiler

The compiler can translate Prolog to Lisp, but that does us no good unless we can conveniently arrange to compile the right Prolog relations and call the right Lisp functions. In other words, we have to integrate the compiler with the `<-` and `?` macros. Surprisingly, we don't need to change these macros at all. Rather, we will change the functions these macros call. When a new clause is entered, we will enter the clause's predicate in the list `*uncompiled*`. This is a one-line addition to `add-clause`:

```
(defvar *uncompiled* nil
  "Prolog symbols that have not been compiled.")

(defun add-clause (clause)
  "Add a clause to the data base, indexed by head's predicate."
  :: The predicate must be a non-variable symbol.
  (let ((pred (predicate (clause-head clause))))
    (assert (and (symbolp pred) (not (variable-p pred))))
    (pushnew pred *db-predicates*)
    (pushnew pred *uncompiled*) ;***"
    (setf (get pred 'clauses)
```

```
(nconc (get-clauses pred) (list clause)))
pred))
```

Now when a query is made, the `?-` macro expands into a call to `top-level-prove`. The list of goals in the query, along with the `show-prolog-vars` goal, is added as the sole clause for the relation `top-level-query`. Next, that query, along with any others that are on the uncompiled list, are compiled. Finally, the newly compiled top-level query function is called.

```
(defun top-level-prove (goals)
  "Prove the list of goals by compiling and calling it."
  ;; First redefine top-level-query
  (clear-predicate 'top-level-query)
  (let ((vars (delete '? (variables-in goals))))
    (add-clause '((top-level-query)
                  ,@goals
                  (show-prolog-vars ,(mapcar #'symbol-name vars)
                                     ,vars))))
  ;; Now run it
  (run-prolog 'top-level-query/0 #'ignore)
  (format t "~&No.")
  (values))

(defun run-prolog (procedure cont)
  "Run a 0-ary prolog procedure with a given continuation."
  ;; First compile anything else that needs it
  (prolog-compile-symbols)
  ;; Reset the trail and the new variable counter
  (setf (fill-pointer *trail*) 0)
  (setf *var-counter* 0)
  ;; Finally, call the query
  (catch 'top-level-prove
    (funcall procedure cont)))

(defun prolog-compile-symbols (&optional (symbols *uncompiled*))
  "Compile a list of Prolog symbols.
  By default, the list is all symbols that need it."
  (mapc #'prolog-compile-symbols)
  (setf *uncompiled* (set-difference *uncompiled* symbols)))

(defun ignore (&rest args)
  (declare (ignore args))
  nil)
```

Note that at the top level, we don't need the continuation to do anything. Arbitrarily, we chose to pass in the function `ignore`, which is defined to ignore its arguments.

This function is useful in a variety of places; some programmers will proclaim it inline and then use a call to `ignore` in place of an `ignore` declaration:

```
(defun third-arg (x y z)
  (ignore x y)
  z)
```

The compiler's calling convention is different from the interpreter, so the primitives need to be redefined. The old definition of the primitive `show-prolog-vars` had three parameters: the list of arguments to the goal, a binding list, and a list of pending goals. The new definition of `show-prolog-vars/2` also has three parameters, but that is just a coincidence. The first two parameters are the two separate arguments to the goal: a list of variable names and a list of variable values. The last parameter is a continuation function. To continue, we call that function, but to fail, we throw to the catch point set up in `top-level-prove`.

```
(defun show-prolog-vars/2 (var-names vars cont)
  "Display the variables, and prompt the user to see
  if we should continue. If not, return to the top level."
  (if (null vars)
      (format t "~&Yes")
      (loop for name in var-names
            for var in vars do
            (format t "~&~a = ~a" name (deref-exp var))))
  (if (continue-p)
      (funcall cont)
      (throw 'top-level-prove nil)))

(defun deref-exp (exp)
  "Build something equivalent to EXP with variables dereferenced."
  (if (atom (deref exp))
      exp
      (reuse-cons
        (deref-exp (first exp))
        (deref-exp (rest exp))
        exp)))
```

With these definitions in place, we can invoke the compiler automatically just by making a query with the `?-` macro.



**Exercise 12.6 [m]** Suppose you define a predicate `p`, which calls `q`, and then define `q`. In some implementations of Lisp, when you make a query like `(?- (p ?x))`, you may get a warning message like “function `q/1 undefined`” before getting the correct

answer. The problem is that each function is compiled separately, so warnings detected during the compilation of p/1 will be printed right away, even if the function q/1 will be defined later. In ANSI Common Lisp there is a way to delay the printing of warnings until a series of compilations are done: wrap the compilation with the macro `with-compilation-unit`. Even if your implementation does not provide this macro, it may provide the same functionality under a different name. Find out if `with-compilation-unit` is already defined in your implementation, or if it can be defined.

## 12.7 Benchmarking the Compiler

Our compiled Prolog code runs the zebra puzzle in 17.4 seconds, a 16-fold speed-up over the interpreted version, for a rate of 740 LIPS.

Another popular benchmark is Lisp's reverse function, which we can code as the `rev` relation:

```
(<- (rev () ()))
(<- (rev (?x . ?a) ?b) (rev ?a ?c) (concat ?c (?x) ?b))

(<- (concat () ?l ?l))
(<- (concat (?x . ?a) ?b (?x . ?c)) (concat ?a ?b ?c))
```

`rev` uses the relation `concat`, which stands for concatenation. (`concat ?a ?b ?c`) is true when `?a` concatenated to `?b` yields `?c`. This relationlike name is preferred over more procedural names like `append`. But `rev` is very similar to the following Lisp definitions:

```
(defun rev (l)
  (if (null l)
      nil
      (app (rev (rest l))
            (list (first l)))))

(defun app (x y)
  (if (null x)
      y
      (cons (first x)
            (app (rest x) y))))
```

Both versions are inefficient. It is possible to write an iterative version of `reverse` that does no extra consing and is tail-recursive:

```
(<- (irev ?l ?r) (irev3 ?l () ?r))
(<- (irev3 (?x . ?l) ?so-far ?r) (irev3 ?l (?x . ?so-far) ?r))
(<- (irev3 () ?r ?r))
```

The Prolog `i rev` is equivalent to this Lisp program:

```
(defun irev (list) (irev2 list nil))

(defun irev2 (list so-far)
  (if (consp list)
      (irev2 (rest list) (cons (first list) so-far))
      so-far))
```

The following table shows times in seconds to execute these routines on lists of length 20 and 100, for both Prolog and Lisp, both interpreted and compiled. (Only compiled Lisp could execute `rev` on a 100-element list without running out of stack space.) Times for the zebra puzzle are also included, although there is no Lisp version of this program.

Problem	Interp. Prolog	Comp. Prolog	Speed-up	Interp. Lisp	Comp. Lisp
zebra	278.000	17.241	16	—	—
rev 20	4.24	.208	20	.241	.0023
rev 100	—	—	—	—	.0614
irev 20	.22	.010	22	.028	.0005
irev 100	9.81	.054	181	.139	.0014

This benchmark is too small to be conclusive, but on these examples the Prolog compiler is 16 to 181 times faster than the Prolog interpreter, slightly faster than interpreted Lisp, but still 17 to 90 times slower than compiled Lisp. This suggests that the Prolog interpreter cannot be used as a practical programming tool, but the Prolog compiler can.

Before moving on, it is interesting to note that Prolog provides for optional arguments automatically. Although there is no special syntax for optional arguments, an often-used convention is to have two versions of a relation, one with  $n$  arguments and one with  $n - 1$ . A single clause for the  $n - 1$  case provides the missing, and therefore "optional," argument. In the following example, `i rev/2` can be considered as a version of `i rev/3` where the missing optional argument is `()`.

```
(<- (irev ?l ?r) (irev ?l () ?r))
(<- (irev (?x . ?l) ?so-far ?r) (irev ?l (?x . ?so-far) ?r))
(<- (irev () ?r ?r))
```

This is roughly equivalent to the following Lisp verison:

---

```
(defun irev (list &optional (so-far nil))
  (if (consp list)
      (irev (rest list) (cons (first list) so-far))
      so-far))
```

## 12.8 Adding More Primitives

Just as a Lisp compiler needs machine instructions to do input/output, arithmetic, and the like, so our Prolog system needs to be able to perform certain primitive actions. For the Prolog interpreter, primitives were implemented by function symbols. When the interpreter went to fetch a list of clauses, if it got a function instead, it called that function, passing it the arguments to the current relation, the current bindings, and a list of unsatisfied goals. For the Prolog compiler, primitives can be installed simply by writing a Lisp function that respects the convention of taking a continuation as the final argument and has a name of the form *symbol/arity*. For example, here's an easy way to handle input and output:

```
(defun read/1 (exp cont)
  (if (unify! exp (read))
      (funcall cont)))

(defun write/1 (exp cont)
  (write (deref-exp exp) :pretty t)
  (funcall cont))
```

Calling (write ?x) will always succeed, so the continuation will always be called. Similarly, one could use (read ?x) to read a value and unify it with ?x. If ?x is unbound, this is the same as assigning the value. However, it is also possible to make a call like (read (?x + ?y)), which succeeds only if the input is a three-element list with + in the middle. It is an easy extension to define read/2 and write/2 as relations that indicate what stream to use. To make this useful, one would need to define open/2 as a relation that takes a pathname as one argument and gives a stream back as the other. Other optional arguments could also be supported, if desired.

The primitive nl outputs a newline:

```
(defun nl/0 (cont) (terpri) (funcall cont))
```

We provided special support for the unification predicate, =. However, we could have simplified the compiler greatly by having a simple definition for =/2:

```
(defun =/2 (?arg1 ?arg2 cont)
  (if (unify! ?arg1 ?arg2)
    (funcall cont)))
```

In fact, if we give our compiler the single clause:

```
(<- (= ?x ?x))
```

it produces just this code for the definition of `=/2`. There are other equality predicates to worry about. The predicate `==/2` is more like `equal` in Lisp. It does no unification, but instead tests if two structures are equal with regard to their elements. A variable is considered equal only to itself. Here's an implementation:

```
(defun ==/2 (?arg1 ?arg2 cont)
  "Are the two arguments EQUAL with no unification,
  but with dereferencing? If so, succeed."
  (if (deref-equal ?arg1 ?arg2)
    (funcall cont)))

(defun deref-equal (x y)
  "Are the two arguments EQUAL with no unification,
  but with dereferencing?"
  (or (eq (deref x) (deref y))
    (and (consp x)
      (consp y)
      (deref-equal (first x) (first y))
      (deref-equal (rest x) (rest y)))))
```

One of the most important primitives is `call`. Like `funcall` in Lisp, `call` allows us to build up a goal and then try to prove it.

```
(defun call/1 (goal cont)
  "Try to prove goal by calling it."
  (deref goal)
  (apply (make-predicate (first goal))
    (length (args goal)))
  (append (args goal) (list cont))))
```

This version of `call` will give a run-time error if the goal is not instantiated to a list whose first element is a properly defined predicate; one might want to check for that, and fail silently if there is no defined predicate. Here's an example of `call` where the goal is legal:

```
> (?- (= ?p member) (call (?p ?x (a b c))))
?P = MEMBER
?X = A;
?P = MEMBER
?X = B;
?P = MEMBER
?X = C;
No.
```

Now that we have `call`, a lot of new things can be implemented. Here are the logical connectives `and` and `or`:

```
(<- (or ?a ?b) (call ?a))
(<- (or ?a ?b) (call ?b))

(<- (and ?a ?b) (call ?a) (call ?b))
```

Note that these are only binary connectives, not the  $n$ -ary special forms used in Lisp. Also, this definition negates most of the advantage of compilation. The goals inside an `and` or `or` will be interpreted by `call`, rather than being compiled.

We can also define `not`, or at least the normal Prolog `not`, which is quite distinct from the logical `not`. In fact, in some dialects, `not` is written `\+`, which is supposed to be reminiscent of the logical symbol  $\nabla$ , that is, “can not be derived.” The interpretation is that if goal  $G$  can not be proved, then  $(\text{not } G)$  is true. Logically, there is a difference between  $(\text{not } G)$  being true and being unknown, but ignoring that difference makes Prolog a more practical programming language. See Lloyd 1987 for more on the formal semantics of negation in Prolog.

Here’s an implementation of `not/1`. Since it has to manipulate the trail, and we may have other predicates that will want to do the same, we’ll package up what was done in `maybe-add-undo-bindings` into the macro `with-undo-bindings`:

```
(defmacro with-undo-bindings (&body body)
  "Undo bindings after each expression in body except the last."
  (if (length=1 body)
      (first body)
      '(let ((old-trail (fill-pointer *trail*)))
         ,(first body)
         ,@(loop for exp in (rest body)
                 collect '(undo-bindings! old-trail)
                 collect exp))))
(defun not/1 (relation cont)
  "Negation by failure: If you can't prove G, then (not G) true."
  ;;= Either way, undo the bindings.
  (with-undo-bindings
    (call/1 relation #'(lambda () (return-from not/1 nil)))
    (funcall cont)))
```

Here's an example where `not` works fine:

```
> (?- (member ?x (a b c)) (not (= ?x b)))
?X = A;
?X = C;
No.
```

Now see what happens when we simply reverse the order of the two goals:

```
> (?- (not (= ?x b)) (member ?x (a b c)))
No.
```

The first example succeeds unless `?x` is bound to `b`. In the second example, `?x` is unbound at the start, so `(= ?x b)` succeeds, the `not` fails, and the `member` goal is never reached. So our implementation of `not` has a consistent procedural interpretation, but it is not equivalent to the declarative interpretation usually given to logical negation. Normally, one would expect that `a` and `c` would be valid solutions to the query, regardless of the order of the goals.

One of the fundamental differences between Prolog and Lisp is that Prolog is relational: you can easily express individual relations. Lisp, on the other hand, is good at expressing collections of things as lists. So far we don't have any way of forming a collection of objects that satisfy a relation in Prolog. We can easily iterate over the objects; we just can't gather them together. The primitive `bagof` is one way of doing the collection. In general, `(bagof ?x (p ?x) ?bag)` unifies `?bag` with a list of all `?x`'s that satisfy `(p ?x)`. If there are no such `?x`'s, then the call to `bagof` fails. A *bag* is an unordered collection with duplicates allowed. For example, the bag  $\{a, b, a\}$  is the same as the bag  $\{a, a, b\}$ , but different from  $\{a, b\}$ . Bags stands in contrast to *sets*, which are unordered collections with no duplicates. The set  $\{a, b\}$  is the same as the set  $\{b, a\}$ . Here is an implementation of `bagof`:

```
(defun bagof/3 (exp goal result cont)
  "Find all solutions to GOAL, and for each solution,
  collect the value of EXP into the list RESULT."
  ;;= Ex: Assume (p 1) (p 2) (p 3). Then:
  ;;=      (bagof ?x (p ?x) ?l) ==> ?l = (1 2 3)
  (let ((answers nil))
    (call/1 goal #'(lambda ()
                      (push (deref-copy exp) answers)))
    (if (and (not (null answers))
              (unify! result (nreverse answers)))
        (funcall cont))))
```

```
(defun deref-copy (exp)
  "Copy the expression, replacing variables with new ones.
  The part without variables can be returned as is."
  (sublis (mapcar #'(lambda (var) (cons (deref var) (?)))
                    (unique-find-anywhere-if #'var-p exp))
          exp))
```

Below we use bagof to collect a list of everyone Sandy likes. Note that the result is a bag, not a set: Sandy appears more than once.

```
> (?- (bagof ?who (likes Sandy ?who) ?bag))
?WHO = SANDY
?BAG = (LEE KIM ROBIN SANDY CATS SANDY);
No.
```

In the next example, we form the bag of every list of length three that has A and B as members:

```
> (?- (bagof ?l (and (length ?l (1+ (1+ (1+ 0))))
                        (and (member a ?l) (member b ?l)))
                     ?bag))
?L = (?5 ?8 ?11 ?68 ?66)
?BAG = ((A B ?17) (A ?21 B) (B A ?31) (?38 A B) (B ?48 A) (?52 B A))
No.
```

Those who are disappointed with a bag containing multiple versions of the same answer may prefer the primitive setof, which does the same computation as bagof but then discards the duplicates.

```
(defun setof/3 (exp goal result cont)
  "Find all unique solutions to GOAL, and for each solution,
  collect the value of EXP into the list RESULT."
  ;;= Ex: Assume (p 1) (p 2) (p 3). Then:
  ;;=      (setof ?x (p ?x) ?l) => ?l = (1 2 3)
  (let ((answers nil))
    (call/1 goal #'(lambda ()
                      (push (deref-copy exp) answers)))
    (if (and (not (null answers))
              (unify! result (delete-duplicates
                           answers
                           :test #'deref-equal)))
        (funcall cont))))
```

Prolog supports arithmetic with the operator is. For example, (is ?x (+ ?y 1)) unifies ?x with the value of ?y plus one. This expression fails if ?y is unbound, and it

gives a run-time error if `?y` is not a number. For our version of Prolog, we can support not just arithmetic but any Lisp expression:

```
(defun is/2 (var exp cont)
  ::= Example: (is ?x (+ 3 (* ?y (+ ?z 4))))
  ::= Or even: (is (?x ?y ?x) (cons (first ?z) ?!))
  (if (and (not (find-if-anywhere #'unbound-var-p exp))
            (unify! var (eval (deref-exp exp)))))
      (funcall cont)))
(defun unbound-var-p (exp)
  "Is EXP an unbound var?"
  (and (var-p exp) (not (bound-p exp))))
```

As an aside, we might as well give the Prolog programmer access to the function `unbound-var-p`. The standard name for this predicate is `var/1`:

```
(defun var/1 (?arg1 cont)
  "Succeeds if ?arg1 is an uninstantiated variable."
  (if (unbound-var-p ?arg1)
      (funcall cont)))
```

The `is` primitive fails if any part of the second argument is unbound. However, there are expressions with variables that can be solved, although not with a direct call to `eval`. For example, the following goal could be solved by binding `?x` to 2:

```
(solve (= 12 (* (+ ?x 1) 4)))
```

We might want to have more direct access to Lisp from Prolog. The problem with `is` is that it requires a check for unbound variables, and it calls `eval` to evaluate arguments recursively. In some cases, we just want to get at Lisp's `apply`, without going through the safety net provided by `is`. The primitive `lisp` does that. Needless to say, `lisp` is not a part of standard Prolog.

```
(defun lisp/2 (?result exp cont)
  "Apply (first exp) to (rest exp), and return the result."
  (if (and (consp (deref exp))
            (unify! ?result (apply (first exp) (rest exp))))
      (funcall cont)))
```



**Exercise 12.7 [m]** Define the primitive `solve/1`, which works like the function `solve` used in `student` (page 225). Decide if it should take a single equation as argument or a list of equations.

- ❑ **Exercise 12.8 [h]** Assume we had a goal of the form (`solve (= 12 (* (+ ?x 1) 4))`). Rather than manipulate the equation when `solve/1` is called at run time, we might prefer to do part of the work at compile time, treating the call as if it were (`solve (= ?x 2)`). Write a Prolog compiler macro for `solve`. Notice that even when you have defined a compiler macro, you still need the underlying primitive, because the predicate might be invoked through a `call/1`. The same thing happens in Lisp: even when you supply a compiler macro, you still need the actual function, in case of a `funcall` or `apply`.
- ❑ **Exercise 12.9 [h]** Which of the predicates `call`, `and`, `or`, `not`, or `repeat` could benefit from compiler macros? Write compiler macros for those predicates that could use one.
- ❑ **Exercise 12.10 [m]** You might have noticed that `call/1` is inefficient in two important ways. First, it calls `make-predicate`, which must build a symbol by appending strings and then look the string up in the Lisp symbol table. Alter `make-predicate` to store the predicate symbol the first time it is created, so it can do a faster lookup on subsequent calls. The second inefficiency is the call to `append`. Change the whole compiler so that the continuation argument comes first, not last, thus eliminating the need for `append` in `call`.
- ❑ **Exercise 12.11 [s]** The primitive `true/0` always succeeds, and `false/0` always fails. Define these primitives. Hint: the first corresponds to a Common Lisp function, and the second is a function already defined in this chapter.
- ❑ **Exercise 12.12 [s]** Would it be possible to write `==/2` as a list of clauses rather than as a primitive?
- ❑ **Exercise 12.13 [m]** Write a version of `deref-copy` that traverses the argument expression only once.

## 12.9 The Cut

In Lisp, it is possible to write programs that backtrack explicitly, although it can be awkward when there are more than one or two backtrack points. In Prolog, backtracking is automatic and implicit, but we don't yet know of any way to *avoid* backtracking. There are two reasons why a Prolog programmer might want to disable backtracking. First, keeping track of the backtrack points takes up time and space. A programmer who knows that a certain problem has only one solution should be able to speed up the computation by telling the program not to consider the other possible branches. Second, sometimes a simple logical specification of a problem will yield redundant solutions, or even some unintended solutions. It may be that simply pruning the search space to eliminate some backtracking will yield only the desired answers, while restructuring the program to give all and only the right answers would be more difficult. Here's an example. Suppose we wanted to define a predicate, `max/3`, which holds when the third argument is the maximum of the first two arguments, where the first two arguments will always be instantiated to numbers. The straightforward definition is:

```
(<- (max ?x ?y ?x) (>= ?x ?y))
(<- (max ?x ?y ?y) (< ?x ?y))
```

Declaratively, this is correct, but procedurally it is a waste of time to compute the `<` relation if the `>=` has succeeded: in that case the `<` can never succeed. The cut symbol, written `!`, can be used to stop the wasteful computation. We could write:

```
(<- (max ?x ?y ?x) (>= ?x ?y) !)
(<- (max ?x ?y ?y))
```

The cut in the first clause says that if the first clause succeeds, then no other clauses will be considered. So now the second clause can not be interpreted on its own. Rather, it is interpreted as "if the first clause fails, then the `max` of two numbers is the second one."

In general, a cut can occur anywhere in the body of a clause, not just at the end. There is no good declarative interpretation of a cut, but the procedural interpretation is two-fold. First, when a cut is "executed" as a goal, it always succeeds. But in addition to succeeding, it sets up a fence that cannot be crossed by subsequent backtracking. The cut serves to cut off backtracking both from goals to the right of the cut (in the same clause) and from clauses below the cut (in the same predicate). Let's look at a more abstract example:

```
(<- (p) (q) (r) ! (s) (t))
(<- (p) (s))
```

In processing the first clause of  $p$ , backtracking can occur freely while attempting to solve  $q$  and  $r$ . Once  $r$  is solved, the cut is encountered. From that point on, backtracking can occur freely while solving  $s$  and  $t$ , but Prolog will never backtrack past the cut into  $r$ , nor will the second clause be considered. On the other hand, if  $q$  or  $r$  failed (before the cut is encountered), then Prolog would go on to the second clause.

Now that the intent of the cut is clear, let's think of how it should be implemented. We'll look at a slightly more complex predicate, one with variables and multiple cuts:

```
(<- (p ?x a) ! (q ?x))
(<- (p ?x b) (r ?x) ! (s ?x))
```

We have to arrange it so that as soon as we backtrack into a cut, no more goals are considered. In the first clause, when  $q/1$  fails, we want to return from  $p/2$  immediately, rather than considering the second clause. Similarly, the first time  $s/1$  fails, we want to return from  $p/2$ , rather than going on to consider other solutions to  $r/1$ . Thus, we want code that looks something like this:

```
(defun p/2 (arg1 arg2 cont)
  (let ((old-trail (fill-pointer *trail*)))
    (if (unify! arg2 'a)
        (progn (q/1 arg1 cont)
               (return-from p/2 nil)))
        (undo-bindings! old-trail)
        (if (unify! arg2 'b)
            (r/1 arg1 #'(lambda ()
                          (progn (s/1 arg1 cont)
                                 (return-from p/2 nil)))))))
```

We can get this code by making a single change to `compile-body`: when the first goal in a body (or what remains of the body) is the cut symbol, then we should generate a `progn` that contains the code for the rest of the body, followed by a `return-from` the predicate being compiled. Unfortunately, the name of the predicate is not available to `compile-body`. We could change `compile-clause` and `compile-body` to take the predicate name as an extra argument, or we could bind the predicate as a special variable in `compile-predicate`. I choose the latter:

```
(defvar *predicate* nil
  "The Prolog predicate currently being compiled")
```

```

(defun compile-predicate (symbol arity clauses)
  "Compile all the clauses for a given symbol/arity
  into a single LISP function."
  (let ((*predicate* (make-predicate symbol arity)) ;***
        (parameters (make-parameters arity)))
    (compile
      (eval
        '(defun ,*predicate* (,@parameters cont) ;***
          ,(maybe-add-undo-bindings
            (mapcar #'(lambda (clause)
                        (compile-clause parameters
                                          clause 'cont))
                    clauses)))))))

(defun compile-body (body cont bindings)
  "Compile the body of a clause."
  (cond
    ((null body)
     '(funcall ,cont))
    ((eq (first body) '!)
     '(progn ,(compile-body (rest body) cont bindings) ;***
              (return-from ,*predicate* nil))) ;***)
    (t (let* ((goal (first body))
              (macro (prolog-compiler-macro (predicate goal)))
              (macro-val (if macro
                            (funcall macro goal (rest body)
                                     cont bindings)))
              (if (and macro (not (eq macro-val :pass)))
                  macro-val
                  ',(make-predicate (predicate goal)
                                    (relation-arity goal))
                  ,(mapcar #'(lambda (arg)
                               (compile-arg arg bindings))
                           (args goal)))
              ,(if (null (rest body))
                  cont
                  '#'(lambda ()
                        ,(compile-body
                          (rest body) cont
                          (bind-new-variables bindings goal)))))))))))

```

-  **Exercise 12.14 [m]** Given the definitions below, figure out what a call to `test-cut` will do, and what it will write:

```

(<- (test-cut) (p a) (p b) ! (p c) (p d))
(<- (test-cut) (p e))

```

```
(<- (p ?x) (write (?x 1)))  
(<- (p ?x) (write (?x 2)))
```

Another way to use the cut is in a *repeat/fail* loop. The predicate *repeat* is defined with the following two clauses:

```
(<- (repeat))  
(<- (repeat) (repeat))
```

An alternate definition as a primitive is:

```
(defun repeat/0 (cont)  
  (loop (funcall cont)))
```

Unfortunately, *repeat* is one of the most abused predicates. Several Prolog books present programs like this:

```
(<- (main)  
    (write "Hello.")  
    (repeat)  
    (write "Command: ")  
    (read ?command)  
    (process ?command)  
    (= ?command exit)  
    (write "Good bye."))
```

The intent is that commands are read one at a time, and then processed. For each command except *exit*, *process* takes the appropriate action and then fails. This causes a backtrack to the *repeat* goal, and a new command is read and processed. When the command is *exit*, the procedure returns.

There are two reasons why this is a poor program. First, it violates the principle of referential transparency. Things that look alike are supposed to be alike, regardless of the context in which they are used. But here there is no way to tell that four of the six goals in the body comprise a loop, and the other goals are outside the loop. Second, it violates the principle of abstraction. A predicate should be understandable as a separate unit. But here the predicate *process* can only be understood by considering the context in which it is called: a context that requires it to fail after processing each command. As Richard O'Keefe 1990 points out, the correct way to write this clause is as follows:

```
(<- (main)
      (write "Hello.")
      (repeat)
        (write "Command: ")
        (read ?command)
        (process ?command)
        (or (= ?command exit) (fail))
      !
      (write "Good bye.))
```

The indentation clearly indicates the limits of the repeat loop. The loop is terminated by an explicit test and is followed by a cut, so that a calling program won't accidentally backtrack into the loop after it has exited. Personally, I prefer a language like Lisp, where the parentheses make constructs like loops explicit and indentation can be done automatically. But O'Keefe shows that well-structured readable programs can be written in Prolog.

The if-then and if-then-else constructions can easily be written as clauses. Note that the if-then-else uses a cut to commit to the then part if the test is satisfied.

```
(<- (if ?test ?then) (if ?then ?else (fail)))
(<- (if ?test ?then ?else)
      (call ?test)
      !
      (call ?then))
(<- (if ?test ?then ?else)
      (call ?else))
```

The cut can be used to implement the nonlogical not. The following two clauses are often given before as the definition of not. Our compiler successfully turns these two clauses into exactly the same code as was given before for the primitive not/1:

```
(<- (not ?p) (call ?p) ! (fail))
(<- (not ?p))
```

## 12.10 “Real” Prolog

The Prolog-In-Lisp system developed in this chapter uses Lisp syntax because it is intended to be embedded in a Lisp system. Other Prolog implementations using Lisp syntax include micro-Prolog, Symbolics Prolog, and LMI Prolog.

However, the majority of Prolog systems use a syntax closer to traditional mathematical notation. The following table compares the syntax of "standard" Prolog to the syntax of Prolog-In-Lisp. While there is currently an international committee working on standardizing Prolog, the final report has not yet been released, so different dialects may have slightly different syntax. However, most implementations follow the notation summarized here. They derive from the Prolog developed at the University of Edinburgh for the DEC-10 by David H. D. Warren and his colleagues. The names for the primitives in the last section are also taken from Edinburgh Prolog.

	Prolog	Prolog-In-Lisp
atom	lower	const
variable	Upper	?var
anonymous	-	?
goal	p(Var,const)	(p ?var const)
rule	p(X) :- q(X).	(<- (p ?x) (q ?x))
fact	p(a).	(<- (p a))
query	?- p(X).	(?- (p ?x))
list	[a,b,c]	(a b c)
cons	[a]Rest	(a . ?rest)
nil	[]	()
and	p(X), q(X)	(and (p ?x) (q ?x))
or	p(X); q(X)	(or (p ?x) (q ?x))
not	\+ p(X)	(not (p ?x))

We have adopted Lisp's bias toward lists; terms are built out of atoms, variables, and conses of other terms. In real Prolog cons cells are provided, but terms are usually built out of *structures*, not lists. The Prolog term p(a,b) corresponds to the Lisp vector #(p/2 a b), not the list (p a b). A minority of Prolog implementations use *structure sharing*. In this approach, every non-atomic term is represented by a skeleton that contains place holders for variables and a header that points to the skeleton and also contains the variables that will fill the place holders. With structure sharing, making a copy is easy: just copy the header, regardless of the size of the skeleton. However, manipulating terms is complicated by the need to keep track of both skeleton and header. See Boyer and Moore 1972 for more on structure sharing.

Another major difference is that real Prolog uses the equivalent of failure continuations, not success continuations. No actual continuation, in the sense of a closure, is built. Instead, when a choice is made, the address of the code for the next choice is pushed on a stack. Upon failure, the next choice is popped off the stack. This is reminiscent of the backtracking approach using Scheme's call/cc facility outlined on page 772.

?

**Exercise 12.15 [m]** Assuming an approach using a stack of failure continuations instead of success continuations, show what the code for `p` and `member` would look like. Note that you need not pass failure continuations around; you can just push them onto a stack that `top-level-prove` will invoke. How would the cut be implemented? Did we make the right choice in implementing our compiler with success continuations, or would failure continuations have been better?

## 12.11 History and References

As described in chapter 11, the idea of logic programming was fairly well understood by the mid-1970s. But because the implementations of that time were slow, logic programming did not catch on. It was the Prolog compiler for the DEC-10 that made logic programming a serious alternative to Lisp and other general-purpose languages. The compiler was developed in 1977 by David H. D. Warren with Fernando Pereira and Luís Pereira. See the paper by Warren (1979) and by all three (1977).

Unfortunately, David H. D. Warren's pioneering work on compiling Prolog has never been published in a widely accessible form. His main contribution was the description of the Warren Abstract Machine (WAM), an instruction set for compiled Prolog. Most existing compilers use this instruction set, or a slight modification of it. This can be done either through byte-code interpretation or through macro-expansion to native machine instructions. Aït-Kaci 1991 provides a good tutorial on the WAM, much less terse than the original (Warren 1983). The compiler presented in this chapter does not use the WAM. Instead, it is modeled after Mark Stickel's (1988) theorem prover. A similar compiler is briefly sketched by Jacques Cohen 1985.

## 12.12 Exercises

?

**Exercise 12.16 [m]** Change the Prolog compiler to allow implicit calls. That is, if a goal is not a cons cell headed by a predicate, compile it as if it were a call. The clause:

```
(<- (p ?x ?y) (?x c) ?y)
```

should be compiled as if it were:

```
(<- (p ?x ?y) (call (?x c)) (call ?y))
```



**Exercise 12.17 [h]** Here are some standard Prolog primitives:

- `get/1` Read a single character and unify it with the argument.
- `put/1` Print a single character.
- `nonvar/1`, `/=`, `/==` The opposites of `var`, `=` and `==`, respectively.
- `integer/1` True if the argument is an integer.
- `atom/1` True if the argument is a symbol (like Lisp's `symbolp`).
- `atomic/1` True if the argument is a number or symbol (like Lisp's `atom`).
- `<`, `>`, `=<`, `>=` Arithmetic comparison; succeeds when the arguments are both instantiated to numbers and the comparison is true.
- `listing/0` Print out the clauses for all defined predicates.
- `listing/1` Print out the clauses for the argument predicate.

Implement these predicates. In each case, decide if the predicate should be implemented as a primitive or a list of clauses, and if it should have a compiler macro.

There are some naming conflicts that need to be resolved. Terms like `atom` have one meaning in Prolog and another in Lisp. Also, in Prolog the normal notation is `\=` and `\==`, not `/=` and `/==`. For Prolog-In-Lisp, you need to decide which notations to use: Prolog's or Lisp's.



**Exercise 12.18 [s]** In Lisp, we are used to writing n-ary calls like `(< 1 n 10)` or `(= x y z)`. Write compiler macros that expand n-ary calls into a series of binary calls. For example, `(< 1 n 10)` should expand into `(and (< 1 n) (< n 10))`.



**Exercise 12.19 [m]** One feature of Lisp that is absent in Prolog is the quote mechanism. Is there a use for quote? If so, implement it; if not, explain why it is not needed.



**Exercise 12.20 [h]** Write a tracing mechanism for Prolog. Add procedures `p-trace` and `p-untrace` to trace and untrace Prolog predicates. Add code to the compiler to generate calls to a printing procedure for goals that are traced. In Lisp, we have to trace procedures when they are called and when they return. In Prolog, there are four cases to consider: the call, successful completion, backtrack into subsequent clauses, and failure with no more clauses. We will call these four cases `call`, `exit`,

redo, and fail, respectively. If we traced member, we would expect tracing output to look something like this:

```
> (?- (member ?x (a b c d)) (fail))
CALL MEMBER: ?1 (A B C D)
EXIT MEMBER: A (A B C D)
REDO MEMBER: ?1 (A B C D)
    CALL MEMBER: ?1 (B C D)
    EXIT MEMBER: B (B C D)
    REDO MEMBER: ?1 (B C D)
        CALL MEMBER: ?1 (C D)
        EXIT MEMBER: C (C D)
        REDO MEMBER: ?1 (C D)
            CALL MEMBER: ?1 (D)
            EXIT MEMBER: D (D)
            REDO MEMBER: ?1 (D)
                CALL MEMBER: ?1 NIL
                REDO MEMBER: ?1 NIL
                FAIL MEMBER: ?1 NIL
                FAIL MEMBER: ?1 (D)
                FAIL MEMBER: ?1 (C D)
                FAIL MEMBER: ?1 (B C D)
                FAIL MEMBER: ?1 (A B C D)
```

No.

**?** **Exercise 12.21 [m]** Some Lisp systems are very slow at compiling functions. KCL is an example; it compiles by translating to C and then calling the C compiler and assembler. In KCL it is best to compile only code that is completely debugged, and run interpreted while developing a program.

Alter the Prolog compiler so that calling the Lisp compiler is optional. In all cases, Prolog functions are translated into Lisp, but they are only compiled to machine language when a variable is set.

**?** **Exercise 12.22 [d]** Some Prolog systems provide the predicate `freeze` to “freeze” a goal until its variables are instantiated. For example, the goal `(freeze x (> x 0))` is interpreted as follows: if `x` is instantiated, then just evaluate the goal `(> x 0)`, and succeed or fail depending on the result. However, if `x` is unbound, then succeed and continue the computation, but remember the goal `(> x 0)` and evaluate it as soon as `x` becomes instantiated. Implement `freeze`.

**?** **Exercise 12.23 [m]** Write a recursive version of `anonymous-variables-in` that does not use a local function.

## 12.13 Answers

**Answer 12.6** Here's a version that works for Texas Instruments and Lucid implementations:

```
(defmacro with-compilation-unit (options &body body)
  "Do the body, but delay compiler warnings until the end."
  ;;; This is defined in Common Lisp the Language, 2nd ed.
  '(,(read-time-case
      #+TI    'compiler:compiler-warnings-context-bind
      #+Lucid 'with-deferred-warnings
      'progn)
    .,body))

(defun prolog-compile-symbols (&optional (symbols *uncompiled*))
  "Compile a list of Prolog symbols.
  By default, the list is all symbols that need it."
  (with-compilation-unit ()
    (mapc #'prolog-compile symbols)
    (setf *uncompiled* (set-difference *uncompiled* symbols))))
```

**Answer 12.9** Macros for and and or are very important, since these are commonly used. The macro for and is trivial:

```
(def-prolog-compiler-macro and (goal body cont bindings)
  (compile-body (append (args goal) body) cont bindings))
```

The macro for or is trickier:

```
(def-prolog-compiler-macro or (goal body cont bindings)
  (let ((disjuncts (args goal)))
    (case (length disjuncts)
      (0 fail)
      (1 (compile-body (cons (first disjuncts) body) cont bindings))
      (t (let ((fn (gensym "F")))
            '(flet ((,fn () ,(compile-body body cont bindings)))
               .,(maybe-add-undo-bindings)
               (loop for g in disjuncts collect
                     (compile-body (list g) '#',fn
                     bindings)))))))
```

**Answer 12.11** true/0 is funcall: when a goal succeeds, we call the continuation. fail/0 is ignore: when a goal fails, we ignore the continuation. We could also define compiler macros for these primitives:

```
(def-prolog-compiler-macro true (goal body cont bindings)
  (compile-body body cont bindings))

(def-prolog-compiler-macro fail (goal body cont bindings)
  (declare (ignore goal body cont bindings))
  nil)
```

### Answer 12.13

```
(defun deref-copy (exp)
  "Build a copy of the expression, which may have variables.
  The part without variables can be returned as is."
  (let ((var-alist nil))
    (labels
      ((walk (exp)
             (deref exp)
             (cond ((consp exp)
                     (reuse-cons (walk (first exp))
                                (walk (rest exp))
                                exp))
                   ((var-p exp)
                     (let ((entry (assoc exp var-alist)))
                       (if (not (null entry))
                           (cdr entry)
                           (let ((var-copy (?)))
                             (push (cons exp var-copy) var-alist)
                             var-copy))))
                   (t exp))))
        (walk exp))))
```

**Answer 12.14** In the first clause of `test-cut`, all four calls to `p` will succeed via the first clause of `p`. Then backtracking will occur over the calls to `(p c)` and `(p d)`. All four combinations of 1 and 2 succeed. After that, backtracking would normally go back to the call to `(p b)`. But the cut prevents this, and the whole `(test-cut)` goal fails, without ever considering the second clause. Here's the actual output:

```
(?- (test-cut))
(A 1)(B 1)(C 1)(D 1)
Yes;
(D 2)
Yes;
(C 2)(D 1)
Yes;
(D 2)
Yes;
No.
```

**Answer 12.17** For example:

```
(defun >/2 (x y cont)
  (if (and (numberp (deref x)) (numberp (deref y)) (> x y))
      (funcall cont)))

(defun numberp/1 (x cont)
  (if (numberp (deref x))
      (funcall cont)))
```

**Answer 12.19** Lisp uses quote in two ways: to distinguish a symbol from the value of the variable represented by that symbol, and to distinguish a literal list from the value that would be returned by evaluating a function call. The first distinction Prolog makes by a lexical convention: variables begin with a question mark in our Prolog, and they are capitalized in real Prolog. The second distinction is not necessary because Prolog is relational rather than functional. An expression is a goal if it is a member of the body of a clause, and is a literal if it is an argument to a goal.

**Answer 12.20** Hint: Here's how `member` could be augmented with calls to a procedure, `prolog-trace`, which will print information about the four kinds of tracing events:

```
(defun member/2 (?arg1 ?arg2 cont)
  (let ((old-trail (fill-pointer *trail*)))
    (exit-cont #'(lambda ()
                  (prolog-trace 'exit 'member ?arg1 ?arg2 )
                  (funcall cont))))
    (prolog-trace 'call 'member ?arg1 ?arg2)
    (if (unify! ?arg2 (cons ?arg1 (?)))
        (funcall exit-cont))
        (undo-bindings! old-trail)
        (prolog-trace 'redo 'member ?arg1 ?arg2)
        (let ((?rest (?)))
          (if (unify! ?arg2 (cons (?) ?rest))
              (member/2 ?arg1 ?rest exit-cont)))
        (prolog-trace 'fail 'member ?arg1 ?arg2))))
```

The definition of `prolog-trace` is:

```
(defvar *prolog-trace-indent* 0)

(defun prolog-trace (kind predicate &rest args)
  (if (member kind '(call redo))
      (incf *prolog-trace-indent* 3))
  (format t "~&~VT~a ~a:~{ ~a~}" 
          *prolog-trace-indent* kind predicate args)
  (if (member kind '(fail exit))
      (decf *prolog-trace-indent* 3)))
```

**Answer 12.23**

```
(defun anonymous-variables-in (tree)
  "Return a list of all variables that occur only once in tree."
  (values (anon-vars-in tree nil nil)))

(defun anon-vars-in (tree seen-once seen-more)
  "Walk the data structure TREE, returning a list of variables
  seen once, and a list of variables seen more than once."
  (cond
    ((consp tree)
     (multiple-value-bind (new-seen-once new-seen-more)
         (anon-vars-in (first tree) seen-once seen-more)
       (anon-vars-in (rest tree) new-seen-once new-seen-more)))
    ((not (variable-p tree)) (values seen-once seen-more))
    ((member tree seen-once)
     (values (delete tree seen-once) (cons tree seen-more)))
    ((member tree seen-more)
     (values seen-once seen-more)))
    (t (values (cons tree seen-once) seen-more))))
```

## CHAPTER 13

---

# Object-Oriented Programming

The programs in this book cover a wide range of problems. It is only natural that a wide range of programming styles have been introduced to attack these problems. One style not yet covered that has gained popularity in recent years is called *object-oriented programming*. To understand what object-oriented programming entails, we need to place it in the context of other styles.

Historically, the first computer programs were written in an *imperative programming* style. A program was construed as a series of instructions, where each instruction performs some action: changing the value of a memory location, printing a result, and so forth. Assembly language is an example of an imperative language.

As experience (and ambition) grew, programmers looked for ways of controlling the complexity of programs. The invention of subroutines marked the *algorithmic* or *procedural programming* style, a subclass of the imperative style. Subroutines are helpful for two reasons: breaking up the problem into small pieces makes each piece easier to understand, and it also makes it possible to reuse pieces. Examples of procedural languages are FORTRAN, C, Pascal, and Lisp with setf.

Subroutines are still dependent on global state, so they are not completely separate pieces. The use of a large number of global variables has been criticized as a factor that makes it difficult to develop and maintain large programs. To eliminate this problem, the *functional programming* style insists that functions access only the parameters that are passed to them, and always return the same result for the same inputs. Functional programs have the advantage of being mathematically clean—it is easy to prove properties about them. However, some applications are more naturally seen as taking action rather than calculating functional values, and are therefore unnatural to program in a functional style. Examples of functional languages are FP and Lisp without setf.

In contrast to imperative languages are *declarative languages*, which attempt to express “what to do” rather than “how to do it.” One type of declarative programming is *rule-based* programming, where a set of rules states how to transform a problem into a solution. Examples of rule-based systems are ELIZA and STUDENT.

An important kind of declarative programming is *logic programming*, where axioms are used to describe constraints, and computation is done by a constructive proof of a goal. An example of logic language is Prolog.

*Object-oriented programming* is another way to tame the problem of global state. Instead of prohibiting global state (as functional programming does), object-oriented programming breaks up the unruly mass of global state and encapsulates it into small, manageable pieces, or objects. This chapter covers the object-oriented approach.

## 13.1 Object-Oriented Programming

Object-oriented programming turns the world of computing on its side: instead of viewing a program primarily as a set of actions which manipulate objects, it is viewed as a set of objects that are manipulated by actions. The state of each object and the actions that manipulate that state are defined once and for all when the object is created. This can lead to modular, robust systems that are easy to use and extend. It also can make systems correspond more closely to the “real world,” which we humans perceive more easily as being made up of objects rather than actions. Examples of object-oriented languages are Simula, C++, and CLOS, the Common Lisp Object System. This chapter will first introduce object-oriented programming in general, and then concentrate on the Common Lisp Object System.

Many people are promoting object-oriented programming as the solution to the software development problem, but it is hard to get people to agree on just what object-orientation means. Peter Wegner 1987 proposes the following formula as a definition:

$$\text{Object-orientation} = \text{Objects} + \text{Classes} + \text{Inheritance}$$

Briefly, *objects* are modules that encapsulate some data and operations on that data. The idea of *information hiding*—insulating the representation of that data from operations outside of the object—is an important part of this concept. *Classes* are groups of similar objects with identical behavior. Objects are said to be instances of classes. *Inheritance* is a means of defining new classes as variants of existing classes. The new class inherits the behavior of the parent class, and the programmer need only specify how the new class is different.

The object-oriented style brings with it a new vocabulary, which is summarized in the following glossary. Each term will be explained in more detail when it comes up.

*class*: A group of similar objects with identical behavior.

*class variable*: A variable shared by all members of a class.

*delegation*: Passing a message from an object to one of its components.

*generic function*: A function that accepts different types or classes of arguments.

*inheritance*: A means of defining new classes as variants of existing classes.

*instance*: An instance of a class is an object.

*instance variable*: A variable encapsulated within an object.

*message*: A name for an action. Equivalent to generic function.

*method*: A means of handling a message for a particular class.

*mymethod*: A method that depends on more than one argument.

*multiple inheritance*: Inheritance from more than one parent class.

*object*: An encapsulation of local state and behavior.

## 13.2 Objects

Object-oriented programming, by definition, is concerned with *objects*. Any datum that can be stored in computer memory can be thought of as an object. Thus, the number 3, the atom x, and the string "hello" are all objects. Usually, however, the term *object* is used to denote a more complex object, as we shall see.

Of course, all programming is concerned with objects, and with procedures operating on those objects. Writing a program to solve a particular problem will necessarily involve writing definitions for both objects and procedures. What distinguishes object-oriented programming is that the primary way of decomposing the problem into modules is based on the objects rather than on the procedures. The difference can best be seen with an example. Here is a simple program to create bank accounts and keep track of withdrawals, deposits, and accumulation of interest. First, the program is written in traditional procedural style:

```
(defstruct account  
  (name "") (balance 0.00) (interest-rate .06))
```

```
(defun account-withdraw (account amt)
  "Make a withdrawal from this account."
  (if (<= amt (account-balance account))
      (decf (account-balance account) amt)
      'insufficient-funds))

(defun account-deposit (account amt)
  "Make a deposit to this account."
  (incf (account-balance account) amt))

(defun account-interest (account)
  "Accumulate interest in this account."
  (incf (account-balance account)
        (* (account-interest-rate account)
           (account-balance account)))))
```

We can create new bank accounts with `make-account` and modify them with `account-withdraw`, `account-deposit`, and `account-interest`. This is a simple problem, and this simple solution suffices. Problems appear when we change the specification of the problem, or when we envision ways that this implementation could be inadvertently used in error. For example, suppose a programmer looks at the account structure and decides to use `(decf (account-balance account))` directly instead of going through the `account-withdraw` function. This could lead to negative account balances, which were not intended. Or suppose that we want to create a new kind of account, where only a certain maximum amount can be withdrawn at one time. There would be no way to ensure that `account-withdraw` would not be applied to this new, limited account.

The problem is that once we have created an account, we have no control over what actions are applied to it. The object-oriented style is designed to provide that control. Here is the same program written in object-oriented style (using plain Lisp):

The function `new-account` creates account objects, which are implemented as closures that encapsulate three variables: the name, balance, and interest rate of the account. An account object also encapsulates functions to handle the five messages to which the object can respond. An account object can do only one thing: receive a message and return the appropriate function to execute that message. For example, if you pass the message `withdraw` to an account object, it will return a function that, when applied to a single argument (the amount to withdraw), will perform the withdrawal action. This function is called the *method* that implements the message. The advantage of this approach is that account objects are completely encapsulated; the information corresponding to the name, balance, and interest rate is only accessible through the five messages. We have a guarantee that no other code can manipulate the information in the account in any other way.<sup>1</sup>

The function `get-method` finds the method that implements a message for a given object. The function `send` gets the method and applies it to a list of arguments. The name `send` comes from the Flavors object-oriented system, which is discussed in the history section (page 456).

```
(defun get-method (object message)
  "Return the method that implements message for this object."
  (funcall object message))

(defun send (object message &rest args)
  "Get the function to implement the message,
  and apply the function to the args."
  (apply (get-method object message) args))
```

Here is an example of the use of `new-account` and `send`:

```
> (setf acct (new-account "J. Random Customer" 1000.00)) =>
#<CLOSURE 23652465>

> (send acct 'withdraw 500.00) => 500.0

> (send acct 'deposit 123.45) => 623.45

> (send acct 'name) => "J. Random Customer"

> (send acct 'balance) => 623.45
```

---

<sup>1</sup>More accurately, we have a guarantee that there is no way to get at the inside of a closure using portable Common Lisp code. Particular implementations may provide debugging tools for getting at this hidden information, such as `inspect`. So closures are not perfect at hiding information from these tools. Of course, no information-hiding method will be guaranteed against such covert channels—even with the most sophisticated software security measures, it is always possible to, say, wipe a magnet over the computer's disks and alter sensitive data.

## 13.3 Generic Functions

The send syntax is awkward, as it is different from the normal Lisp function-calling syntax, and it doesn't fit in with the other Lisp tools. For example, we might like to say (mapcar 'balance accounts), but with messages we would have to write that as:

```
(mapcar #'(lambda (acct) (send acct 'balance)) accounts)
```

We can fix this problem by defining *generic* functions that find the right method to execute a message. For example, we could define:

```
(defun withdraw (object &rest args)
  "Define withdraw as a generic function on objects."
  (apply (get-method object 'withdraw) args))
```

and then write (withdraw acct x) instead of (send acct 'withdraw x). The function withdraw is generic because it not only works on account objects but also works on any other class of object that handles the withdraw message. For example, we might have a totally unrelated class, army, which also implements a withdraw method. Then we could say (send 5th-army 'withdraw) or (withdraw 5th-army) and have the correct method executed. So object-oriented programming eliminates many problems with name clashes that arise in conventional programs.

Many of the built-in Common Lisp functions can be considered generic functions, in that they operate on different types of data. For example, sqrt does one thing when passed an integer and quite another when passed an imaginary number. The sequence functions (like find or delete) operate on lists, vectors, or strings. These functions are not implemented like withdraw, but they still act like generic functions.<sup>2</sup>

## 13.4 Classes

It is possible to write macros to make the object-oriented style easier to read and write. The macro define-class defines a class with its associated message-handling methods. It also defines a generic function for each message. Finally, it allows the programmer to make a distinction between variables that are associated with each object and those that are associated with a class and are shared by all members of the class. For example, you might want to have all instances of the class account share the same interest rate, but you wouldn't want them to share the same balance.

---

<sup>2</sup>There is a technical sense of “generic function” that is used within CLOS. These functions are not generic according to this technical sense.

```
(defmacro define-class (class inst-vars class-vars &body methods)
  "Define a class for object-oriented programming."
  ;;= Define constructor and generic functions for methods
  '(let ,class-args
    (mapcar #'ensure-generic-fn ,(mapcar #'first methods))
    (defun ,class ,inst-args
      #'(lambda (message)
          (case message
            ,@(mapcar #'make-clause methods)))))

  (defun make-clause (clause)
    "Translate a message from define-class into a case clause."
    '(.,(first clause) #'(lambda ,(second clause) .,(rest2 clause)))))

  (defun ensure-generic-fn (message)
    "Define an object-oriented dispatch function for a message,
     unless it has already been defined as one."
    (unless (generic-fn-p message)
      (let ((fn #'(lambda (object &rest args)
                    (apply (get-method object message) args))))
        (setf (symbol-function message) fn)
        (setf (get message 'generic-fn) fn)))))

  (defun generic-fn-p (fn-name)
    "Is this a generic function?"
    (and (fboundp fn-name)
         (eq (get fn-name 'generic-fn) (symbol-function fn-name))))
```

Now we define the class account with this macro. We make interest-rate a class variable, one that is shared by all accounts:

```
(define-class account (name &optional (balance 0.00))
  ((interest-rate .06))
  (withdraw (amt) (if (<= amt balance)
                      (decf balance amt)
                      'insufficient-funds)))
  (deposit (amt) (incf balance amt))
  (balance () balance)
  (name () name)
  (interest () (incf balance (* interest-rate balance))))
```

Here we use the generic functions defined by this macro:

```
> (setf acct2 (account "A. User" 2000.00)) => #<CLOSURE 24003064>
> (deposit acct2 42.00) => 2042.0
> (interest acct2) => 2164.52
```

```
> (balance acct2) => 2164.52
> (balance acct) => 623.45
```

In this last line, the generic function `balance` is applied to `acct`, an object that was created before we even defined the account class and the function `balance`. But `balance` still works properly on this object, because it obeys the message-passing protocol.

## 13.5 Delegation

Suppose we want to create a new kind of account, one that requires a password for each action. We can define a new class, `password-account`, that has two message clauses. The first clause allows for changing the password (if you have the original password), and the second is an `otherwise` clause, which checks the password given and, if it is correct, passes the rest of the arguments on to the account that is being protected by the password.

The definition of `password-account` takes advantage of the internal details of `define-class` in two ways: it makes use of the fact that `otherwise` can be used as a catch-all clause in a case form, and it makes use of the fact that the dispatch variable is called `message`. Usually, it is not a good idea to rely on details about the implementation of a macro, and soon we will see cleaner ways of defining classes. But for now, this simple approach works:

```
(define-class password-account (password acct) ()
  (change-password (pass new-pass)
    (if (equal pass password)
        (setf password new-pass)
        'wrong-password))
  (otherwise (pass &rest args)
    (if (equal pass password)
        (apply message acct args)
        'wrong-password)))
```

Now we see how the class `password-account` can be used to provide protection for an existing account:

```
(setf acct3 (password-account "secret" acct2)) => #<CLOSURE 33427277>
> (balance acct3 "secret") => 2164.52
> (withdraw acct3 "guess" 2000.00) => WRONG-PASSWORD
> (withdraw acct3 "secret" 2000.00) => 164.52
```

Now let's try one more example. Suppose we want to have a new class of account

where only a limited amount of money can be withdrawn at any time. We could define the class `limited-account`:

```
(define-class limited-account (limit acct) ()
  (withdraw (amt)
    (if (> amt limit)
        'over-limit
        (withdraw acct amt)))
  (otherwise (&rest args)
    (apply message acct args)))
```

This definition redefines the `withdraw` message to check if the limit is exceeded before passing on the message, and it uses the `otherwise` clause simply to pass on all other messages unchanged. In the following example, we set up an account with both a password and a limit:

```
> (setf acct4 (password-account "pass"
                                (limited-account 100.00
                                                (account "A. Thrifty Spender" 500.00)))) =>
#<CLOSURE 34136775>

> (withdraw acct4 "pass" 200.00) => OVER-LIMIT
> (withdraw acct4 "pass" 20.00) => 480.0
> (withdraw acct4 "guess" 20.00) => WRONG-PASSWORD
```

Note that functions like `withdraw` are still simple generic functions that just find the right method and apply it to the arguments. The trick is that each class defines a different way to handle the `withdraw` message. Calling `withdraw` with `acct4` as argument results in the following flow of control. First, the method in the `password-account` class checks that the password is correct. If it is, it calls the method from the `limited-account` class. If the limit is not exceeded, we finally call the method from the `account` class, which decrements the balance. Passing control to the method of a component is called *delegation*.

The advantage of the object-oriented style is that we can introduce a new class by writing one definition that is localized and does not require changing any existing code. If we had written this in traditional procedural style, we would end up with functions like the following:

```
(defun withdraw (acct amt &optional pass)
  (cond ((and (typep acct 'password-account)
              (not (equal pass (account-password acct))))
         'wrong-password)
        ((and (typep acct 'limited-account)
```

```

        (> amt (account-limit account)))
  'over-limit)
((> amt balance)
 'insufficient-funds)
(t (decf balance amt)))

```

There is nothing wrong with this, as an individual function. The problem is that when the bank decides to offer a new kind of account, we will have to change this function, along with all the other functions that implement actions. The “definition” of the new account is scattered rather than localized, and altering a bunch of existing functions is usually more error prone than writing a new class definition.

## 13.6 Inheritance

In the following table, data types (classes) are listed across the horizontal axis, and functions (messages) are listed up and down the vertical axis. A complete program needs to fill in all the boxes, but the question is how to organize the process of filling them in. In the traditional procedural style, we write function definitions that fill in a row at a time. In the object-oriented style, we write class definitions that fill in a column at a time. A third style, the *data-driven* or *generic* style, fills in only one box at a time.

	account	limited-account	password-account	...
name			<i>object</i>	
deposit			<i>oriented</i>	
withdraw	<i>function</i>	<i>oriented</i>		
balance				
interest	<i>generic</i>			
...				

In this table there is no particular organization to either axis; both messages and classes are listed in random order. This ignores the fact that classes are organized hierarchically: both limited-account and password-account are subclasses of account. This was implicit in the definition of the classes, because both limited-account and password-account contain accounts as components and delegate messages to those components. But it would be cleaner to make this relationship explicit.

The defstruct mechanism does allow for just this kind of explicit inheritance. If we had defined account as a structure, then we could define limited-account with:

```
(defstruct (limited-account (:include account)) limit)
```

Two things are needed to provide an inheritance facility for classes. First, we should modify `define-class` so that it takes the name of the class to inherit from as the second argument. This will signal that the new class will inherit all the instance variables, class variables, and methods from the parent class. The new class can, of course, define new variables and methods, or it can shadow the parent's variables and methods. In the form below, we define `limited-account` to be a subclass of `account` that adds a new instance variable, `limit`, and redefines the `withdraw` method so that it checks for amounts that are over the limit. If the amount is acceptable, then it uses the function `call-next-method` (not yet defined) to get at the `withdraw` method for the parent class, `account`.

```
(define-class limited-account account (limit) ()
  (withdraw (amt)
    (if (> amt limit)
      'over-limit
      (call-next-method))))
```

If inheritance is a good thing, then multiple inheritance is an even better thing. For example, assuming we have defined the classes `limited-account` and `password-account`, it is very convenient to define the following class, which inherits from both of them:

```
(define-class limited-account-with-password
  (password-account limited-account))
```

Notice that this new class adds no new variables or methods. All it does is combine the functionality of two parent classes into one.

- ❑ **Exercise 13.1 [d]** Define a version of `define-class` that handles inheritance and `call-next-method`.
- ❑ **Exercise 13.2 [d]** Define a version of `define-class` that handles multiple inheritance.

## 13.7 CLOS: The Common Lisp Object System

So far, we have developed an object-oriented programming system using a macro, `define-class`, and a protocol for implementing objects as closures. There have been many proposals for adding object-oriented features to Lisp, some similar to our approach, some quite different. Recently, one approach has been approved to become an official part of Common Lisp, so we will abandon our ad hoc approach and devote the rest of this chapter to CLOS, the Common Lisp Object System. The correspondence between our system and CLOS is summarized here:

our system	CLOS
<code>define-class</code>	<code>defclass</code>
<i>methods defined in class</i>	<code>defmethod</code>
<i>class-name</i>	<code>make-instance</code>
<code>call-next-method</code>	<code>call-next-method</code>
<code>ensure-generic-fn</code>	<code>ensure-generic-function</code>

Like most object-oriented systems, CLOS is primarily concerned with defining classes and methods for them, and in creating instances of the classes. In CLOS the macro `defclass` defines a class, `defmethod` defines a method, and `make-instance` creates an instance of a class—an object. The general form of the macro `defclass` is:

```
(defclass class-name (superclass...) (slot-specifier...) optional-class-option...)
```

The class-options are rarely used. `defclass` can be used to define the class `account`:

```
(defclass account ()
  ((name :initarg :name :reader name)
   (balance :initarg :balance :initform 0.00 :accessor balance)
   (interest-rate :allocation :class :initform .06
      :reader interest-rate)))
```

In the definition of `account`, we see that the list of superclasses is empty, because `account` does not inherit from any classes. There are three slot specifiers, for the `name`, `balance`, and `interest-rate` slots. Each slot name can be followed by optional keyword/value pairs defining how the slot is used. The `name` slot has an `:initarg` option, which says that the `name` can be specified when a new `account` is created with `make-instance`. The `:reader` slot creates a method called `name` to get at the current value of the slot.

The `balance` slot has three options: another `:initarg`, saying that the `balance` can be specified when a new `account` is made; an `:initform`, which says that if the `balance` is not specified, it defaults to `0.00`, and an `:accessor`, which creates a

method for getting at the slot's value just as :reader does, and also creates a method for updating the slot with setf.

The interest-rate slot has an :initform option to give it a default value and an :allocation option to say that this slot is part of the class, not of each instance of the class.

Here we see the creation of an object, and the application of the automatically defined methods to it.

```
> (setf a1 (make-instance 'account :balance 5000.00
                           :name "Fred")) => #<ACCOUNT 26726272>
> (name a1) => "Fred"
> (balance a1) => 5000.0
> (interest-rate a1) => 0.06
```

CLOS differs from most object-oriented systems in that methods are defined separately from classes. To define a method (besides the ones defined automatically by :reader, :writer, or :accessor options) we use the defmethod macro. It is similar to defun in form:

```
(defmethod method-name (parameter...) body...)
```

Required parameters to a defmethod can be of the form (*var class*), meaning that this is a method that applies only to arguments of that class. Here is the method for withdrawing from an account. Note that CLOS does not have a notion of instance variable, only instance slot. So we have to use the method (balance acct) rather than the instance variable balance:

```
(defmethod withdraw ((acct account) amt)
  (if (< amt (balance acct))
      (decf (balance acct) amt)
      'insufficient-funds))
```

With CLOS it is easy to define a limited-account as a subclass of account, and to define the withdraw method for limited-accounts:

```
(defclass limited-account (account)
  ((limit :initarg :limit :reader limit)))
(defmethod withdraw ((acct limited-account) amt)
  (if (> amt (limit acct))
      'over-limit
      (call-next-method)))
```

Note the use of `call-next-method` to invoke the `withdraw` method for the `account` class. Also note that all the other methods for accounts automatically work on instances of the class `limited-account`, because it is defined to inherit from `account`. In the following example, we show that the `name` method is inherited, that the `withdraw` method for `limited-account` is invoked first, and that the `withdraw` method for `account` is invoked by the `call-next-method` function:

```
> (setf a2 (make-instance 'limited-account
                           :name "A. Thrifty Spender"
                           :balance 500.00 :limit 100.00)) =>
#<LIMITED-ACCOUNT 24155343>
> (name a2) => "A. Thrifty Spender"
> (withdraw a2 200.00) => OVER-LIMIT
> (withdraw a2 20.00) => 480.0
```

In general, there may be several methods appropriate to a given message. In that case, all the appropriate methods are gathered together and sorted, most specific first. The most specific method is then called. That is why the method for `limited-account` is called first rather than the method for `account`. The function `call-next-method` can be used within the body of a method to call the next most specific method.

The complete story is actually even more complicated than this. As one example of the complication, consider the class `audited-account`, which prints and keeps a trail of all deposits and withdrawals. It could be defined as follows using a new feature of CLOS, `:before` and `:after` methods:

```
(defclass audited-account (account)
  ((audit-trail :initform nil :accessor audit-trail)))
(defmethod withdraw :before ((acct audited-account) amt)
  (push (print '(withdrawn ,amt))
        (audit-trail acct)))
(defmethod withdraw :after ((acct audited-account) amt)
  (push (print '(withdrawal ,amt) done))
        (audit-trail acct)))
```

Now a call to `withdraw` with a `audited-account` as the first argument yields three applicable methods: the primary method from `account` and the `:before` and `:after` methods. In general, there might be several of each kind of method. In that case, all the `:before` methods are called in order, most specific first. Then the most specific primary method is called. It may choose to invoke `call-next-method` to get at the other methods. (It is an error for a `:before` or `:after` method to use `call-next-method`.) Finally, all the `:after` methods are called, least specific first.

The values from the :before and :after methods are ignored, and the value from the primary method is returned. Here is an example:

```
> (setf a3 (make-instance 'audited-account :balance 1000.00))
#<AUDITED-ACCOUNT 33555607>

> (withdraw a3 100.00)
(WITHDRAWING 100.0)
(WITHDRAWAL (100.0) DONE)
900.0

> (audit-trail a3)
((WITHDRAWAL (100.0) DONE) (WITHDRAWING 100.0))

> (setf (audit-trail a3) nil)
NIL
```

The last interaction shows the biggest flaw in CLOS: it fails to encapsulate information. In order to make the audit-trail accessible to the withdraw methods, we had to give it accessor methods. We would like to encapsulate the writer function for audit-trail so that it can only be used with deposit and withdraw. But once the writer function is defined it can be used anywhere, so an unscrupulous outsider can destroy the audit trail, setting it to nil or anything else.

## 13.8 A CLOS Example: Searching Tools

CLOS is most appropriate whenever there are several types that share related behavior. A good example of an application that fits this description is the set of searching tools defined in section 6.4. There we defined functions for breadth-first, depth-first, and best-first search, as well as tree- and graph-based search. We also defined functions to search in particular domains, such as planning a route between cities.

If we had written the tools in a straightforward procedural style, we would have ended up with dozens of similar functions. Instead, we used higher-order functions to control the complexity. In this section, we see how CLOS can be used to break up the complexity in a slightly different fashion.

We begin by defining the class of search problems. Problems will be classified according to their domain (route planning, etc.), their topology (tree or graph) and their search strategy (breadth-first or depth-first, etc.). Each combination of these features results in a new class of problem. This makes it easy for the user to add a new class to represent a new domain, or a new search strategy. The basic class, `problem`, contains a single-instance variable to hold the unexplored states of the problem.

---

```
(defclass problem ()
  ((states :initarg :states :accessor problem-states)))
```

The function `searcher` is similar to the function `tree-search` of section 6.4. The main difference is that `searcher` uses generic functions instead of passing around functional arguments.

```
(defmethod searcher ((prob problem))
  "Find a state that solves the search problem."
  (cond ((no-states-p prob) fail)
        ((goal-p prob) (current-state prob))
        (t (let ((current (pop-state prob)))
              (setf (problem-states prob)
                    (problem-combiner
                     prob
                     (problem-successors prob current)
                     (problem-states prob))))
            (searcher prob))))
```

`searcher` does not assume that the problem states are organized in a list; rather, it uses the generic function `no-states-p` to test if there are any states, `pop-state` to remove and return the first state, and `current-state` to access the first state. For the basic `problem` class, we will in fact implement the states as a list, but another class of problem is free to use another representation.

```
(defmethod current-state ((prob problem))
  "The current state is the first of the possible states."
  (first (problem-states prob)))

(defmethod pop-state ((prob problem))
  "Remove and return the current state."
  (pop (problem-states prob)))

(defmethod no-states-p ((prob problem))
  "Are there any more unexplored states?"
  (null (problem-states prob)))
```

In `tree-search`, we included a statement to print debugging information. We can do that here, too, but we can hide it in a separate method so as not to clutter up the main definition of `searcher`. It is a `:before` method because we want to see the output before carrying out the operation.

```
(defmethod searcher :before ((prob problem))
  (dbg 'search "~~&; Search: ~a" (problem-states prob)))
```

The generic functions that remain to be defined are `goal-p`, `problem-combiner`, and `problem-successors`. We will address `goal-p` first, by recognizing that for many problems we will be searching for a state that is `eql` to a specified goal state. We define the class `eql-problem` to refer to such problems, and specify `goal-p` for that class. Note that we make it possible to specify the goal when a problem is created, but not to change the goal:

```
(defclass eql-problem (problem)
  ((goal :initarg :goal :reader problem-goal)))

(defmethod goal-p ((prob eql-problem))
  (eql (current-state prob) (problem-goal prob)))
```

Now we are ready to specify two search strategies: depth-first search and breadth-first search. We define problem classes for each strategy and specify the `problem-combiner` function:

```
(defclass dfs-problem (problem) ()
  (:documentation "Depth-first search problem."))

(defclass bfs-problem (problem) ()
  (:documentation "Breadth-first search problem.))

(defmethod problem-combiner ((prob dfs-problem) new old)
  "Depth-first search looks at new states first."
  (append new old))

(defmethod problem-combiner ((prob bfs-problem) new old)
  "Depth-first search looks at old states first."
  (append old new))
```

While this code will be sufficient for our purposes, it is less than ideal, because it breaks an information-hiding barrier. It treats the set of old states as a list, which is the default for the `problem` class but is not necessarily the implementation that every class will use. It would have been cleaner to define generic functions `add-states-to-end` and `add-states-to-front` and then define them with `append` in the default class. But Lisp provides such nice list-manipulation primitives that it is difficult to avoid the temptation of using them directly.

Of course, the user who defines a new implementation for `problem-states` could just redefine `problem-combiner` for the offending classes, but this is precisely what object-oriented programming is designed to avoid: specializing one abstraction (`states`) should not force us to change anything in another abstraction (`search strategy`).

The last step is to define a class that represents a particular domain, and define problem-successors for that domain. As the first example, consider the simple binary tree search from section 6.4. Naturally, this gets represented as a class:

```
(defclass binary-tree-problem (problem) ())
  (defmethod problem-successors ((prob binary-tree-problem) state)
    (let ((n (* 2 state)))
      (list n (+ n 1))))
```

Now suppose we want to solve a binary-tree problem with breadth-first search, searching for a particular goal. Simply create a class that mixes in `binary-tree-problem`, `eql-problem` and `bfs-problem`, create an instance of that class, and call `searcher` on that instance:

```
(defclass binary-tree-eql-bfs-problem
  (binary-tree-problem eql-problem bfs-problem) ())
> (setf p1 (make-instance 'binary-tree-eql-bfs-problem
                           :states '(1) :goal 12))
#<BINNARY-TREE-EQL-BFS-PROBLEM 26725536>
> (searcher p1)
;; Search: (1)
;; Search: (2 3)
;; Search: (3 4 5)
;; Search: (4 5 6 7)
;; Search: (5 6 7 8 9)
;; Search: (6 7 8 9 10 11)
;; Search: (7 8 9 10 11 12 13)
;; Search: (8 9 10 11 12 13 14 15)
;; Search: (9 10 11 12 13 14 15 16 17)
;; Search: (10 11 12 13 14 15 16 17 18 19)
;; Search: (11 12 13 14 15 16 17 18 19 20 21)
;; Search: (12 13 14 15 16 17 18 19 20 21 22 23)
12
```

### *Best-First Search*

It should be clear how to proceed to define best-first search: define a class to represent best-first search problems, and then define the necessary methods for that class. Since the search strategy only affects the order in which states are explored, the only method necessary will be for `problem-combiner`.

```
(defclass best-problem (problem) ()
  (:documentation "A Best-first search problem."))
(defmethod problem-combiner ((prob best-problem) new old)
  "Best-first search sorts new and old according to cost-fn."
  (sort (append new old) #'<
    :key #'(lambda (state) (cost-fn prob state))))
```

This introduces the new function `cost-fn`; naturally it will be a generic function. The following is a `cost-fn` that is reasonable for any `eql-problem` dealing with numbers, but it is expected that most domains will specialize this function.

```
(defmethod cost-fn ((prob eql-problem) state)
  (abs (- state (problem-goal prob))))
```

Beam search is a modification of best-first search where all but the best  $b$  states are thrown away on each iteration. A beam search problem is represented by a class where the instance variable `beam-width` holds the parameter  $b$ . If this nil, then full best-first search is done. Beam search is implemented by an `:around` method on `problem-combiner`. It calls the next method to get the list of states produced by best-first search, and then extracts the first  $b$  elements.

```
(defclass beam-problem (problem)
  ((beam-width :initarg :beam-width :initform nil
    :reader problem-beam-width)))
(defmethod problem-combiner :around ((prob beam-problem) new old)
  (let ((combined (call-next-method)))
    (subseq combined 0 (min (problem-beam-width prob)
      (length combined)))))
```

Now we apply beam search to the binary-tree problem. As usual, we have to make up another class to represent this type of problem:

```
(defclass binary-tree-eql-best-beam-problem
  (binary-tree-problem eql-problem best-problem beam-problem)
  ())
> (setf p3 (make-instance 'binary-tree-eql-best-beam-problem
                           :states '(1) :goal 12 :beam-width 3))
#<BINNARY-TREE-EQL-BEST-BEAM-PROBLEM 27523251>
> (searcher p3)
;; Search: (1)
;; Search: (3 2)
;; Search: (7 6 2)
;; Search: (14 15 6)
;; Search: (15 6 28)
```

```
;; Search: (6 28 30)
;; Search: (12 13 28)
12
```

So far the case for CLOS has not been compelling. The code in this section duplicates the functionality of code in section 6.4, but the CLOS code tends to be more verbose, and it is somewhat disturbing that we had to make up so many long class names. However, this verbosity leads to flexibility, and it is easier to extend the CLOS code by adding new specialized classes. It is useful to make a distinction between the systems programmer and the applications programmer. The systems programmer would supply a library of classes like `dfs-problem` and generic functions like `searcher`. The applications programmer then just picks what is needed from the library. From the following we see that it is not too difficult to pick out the right code to define a trip-planning searcher. Compare this with the definition of `trip` on page 198 to see if you prefer CLOS in this case. The main difference is that here we say that the cost function is `air-distance` and the successors are the `neighbors` by defining methods; in `trip` we did it by passing parameters. The latter is a little more succinct, but the former may be more clear, especially as the number of parameters grows.

```
(defclass trip-problem (binary-tree-eql-best-beam-problem)
  ((beam-width :initform 1)))

(defmethod cost-fn ((prob trip-problem) city)
  (air-distance (problem-goal prob) city))

(defmethod problem-successors ((prob trip-problem) city)
  (neighbors city))
```

With the definitions in place, it is easy to use the searching tool:

```
> (setf p4 (make-instance 'trip-problem
                           :states (list (city 'new-york))
                           :goal (city 'san-francisco)))
#<TRIP-PROBLEM 31572426>

> (searcher p4)
;; Search: ((NEW-YORK 73.58 40.47))
;; Search: ((PITTSBURG 79.57 40.27))
;; Search: ((CHICAGO 87.37 41.5))
;; Search: ((KANSAS-CITY 94.35 39.06))
;; Search: ((DENVER 105.0 39.45))
;; Search: ((FLAGSTAFF 111.41 35.13))
;; Search: ((RENO 119.49 39.3))
;; Search: ((SAN-FRANCISCO 122.26 37.47))
(SAN-FRANCISCO 122.26 37.47)
```

## 13.9 Is CLOS Object-Oriented?

There is some argument whether CLOS is really object-oriented at all. The arguments are:

CLOS is an object-oriented system because it provides all three of the main criteria for object-orientation: objects with internal state, classes of objects with specialized behavior for each class, and inheritance between classes.

CLOS is *not* an object-oriented system because it does not provide modular objects with information-hiding. In the audited-account example, we would like to encapsulate the audit-trail instance variable so that only the withdraw methods can change it. But because methods are written separately from class definitions, we could not do that. Instead, we had to define an accessor for audit-trail. That enabled us to write the withdraw methods, but it also made it possible for anyone else to alter the audit trail as well.

CLOS is *more general* than an object-oriented system because it allows for methods that specialize on more than one argument. In true object-oriented systems, methods are associated with objects of a particular class. This association is lexically obvious (and the message-passing metaphor is clear) when we write the methods inside the definition of the class, as in our define-class macro. The message-passing metaphor is still apparent when we write generic functions that dispatch on the class of their first argument, which is how we've been using CLOS so far.

But CLOS methods can dispatch on the class of any required argument, or any combination of them. Consider the following definition of conc, which is like append except that it works for vectors as well as lists. Rather than writing conc using conditional statements, we can use the multimethod dispatch capabilities of CLOS to define the four cases: (1) the first argument is nil, (2) the second argument is nil, (3) both arguments are lists, and (4) both arguments are vectors. Notice that if one of the arguments is nil there will be two applicable methods, but the method for null will be used because the class null is more specific than the class list.

```
(defmethod conc ((x null) y) y)
(defmethod conc (x (y null)) x)
(defmethod conc ((x list) (y list))
  (cons (first x) (conc (rest x) y)))
(defmethod conc ((x vector) (y vector))
  (let ((vect (make-array (+ (length x) (length y)))))
    (replace vect x)
    (replace vect y :start1 (length x))))
```

Here we see that this definition works:

```
> (conc nil '(a b c)) => (A B C)
> (conc '(a b c) nil) => (A B C)
> (conc '(a b c) '(d e f)) => (A B C D E F)
> (conc '#(a b c) '#(d e f)) => #(A B C D E F)
```

It works, but one might well ask: where are the objects? The metaphor of passing a message to an object does not apply here, unless we consider the object to be the list of arguments, rather than a single privileged argument.

It is striking that this style of method definition is very similar to the style used in Prolog. As another example, compare the following two definitions of `len`, a relation/function to compute the length of a list:

<code>;; CLOS</code>	<code>%% Prolog</code>
<code>(defmethod len ((x null)) 0)</code>	<code>len([],0).</code>
<code>(defmethod len ((x cons))</code>	<code>len([X L],N1) :-</code>
<code>      (+ 1 (len (rest x))))</code>	<code>      len(L,N), N1 is N+1.</code>

## 13.10 Advantages of Object-Oriented Programming

Bertrand Meyer, in his book on the object-oriented language Eiffel (1988), lists five qualities that contribute to software quality:

- *Correctness*. Clearly, a correct program is of the upmost importance.
- *Robustness*. Programs should continue to function in a reasonable manner even for input that is beyond the original specifications.
- *Extendability*. Programs should be easy to modify when the specifications change.
- *Reusability*. Program components should be easy to transport to new programs, thus amortizing the cost of software development over several projects.
- *Compatibility*. Programs should interface well with other programs. For example, a spreadsheet program should not only manipulate numbers correctly but also be compatible with word processing programs, so that spreadsheets can easily be included in documents.

Here we list how the object-oriented approach in general and CLOS in particular can effect these measures of quality:

- *Correctness.* Correctness is usually achieved in two stages: correctness of individual modules and correctness of the whole system. The object-oriented approach makes it easier to prove correctness for modules, since they are clearly defined, and it may make it easier to analyze interactions between modules, since the interface is strictly limited. CLOS does not provide for information-hiding the way other systems do.
- *Robustness.* Generic functions make it possible for a function to accept, at run time, a class of argument that the programmer did not anticipate at compile time. This is particularly true in CLOS, because multiple inheritance makes it feasible to write default methods that can be used by a wide range of classes.
- *Extendability.* Object-oriented systems with inheritance make it easy to define new classes that are slight variants on existing ones. Again, CLOS's multiple inheritance makes extensions even easier than in single-inheritance systems.
- *Reusability.* This is the area where the object-oriented style makes the biggest contribution. Instead of writing each new program from scratch, object-oriented programmers can look over a library of classes, and either reuse existing classes as is, or specialize an existing class through inheritance. Large libraries of CLOS classes have not emerged yet. Perhaps they will when the language is more established.
- *Compatibility.* The more programs use standard components, the more they will be able to communicate with each other. Thus, an object-oriented program will probably be compatible with other programs developed from the same library of classes.

### 13.11 History and References

The first object-oriented language was Simula, which was designed by Ole-Johan Dahl and Krysten Nygaard (1966, Nygaard and Dahl 1981) as an extension of Algol 60. It is still in use today, mostly in Norway and Sweden. Simula provides the ability to define classes with single inheritance. Methods can be inherited from a superclass or overridden by a subclass. It also provides *coroutines*, class instances that execute continuously, saving local state in instance variables but periodically pausing to let other coroutines run. Although Simula is a general-purpose language, it provides special support for simulation, as the name implies. The built-in class `simulation` allows a programmer to keep track of simulated time while running a set of processes as coroutines.

In 1969 Alan Kay was a graduate student at the University of Utah. He became aware of Simula and realized that the object-oriented style was well suited to his research in graphics (Kay 1969). A few years later, at Xerox, he joined with Adele Goldberg and Daniel Ingalls to develop the Smalltalk language (see Goldberg and Robinson 1983). While Simula can be viewed as an attempt to add object-oriented features to strongly typed Algol 60, Smalltalk can be seen as an attempt to use the dynamic, loosely typed features of Lisp, but with methods and objects replacing functions and s-expressions. In Simula, objects existed alongside traditional data types like numbers and strings; in Smalltalk, every datum is an object. This gave Smalltalk the feel of an integrated Lisp environment, where the user can inspect, copy, or edit any part of the environment. In fact, it was not the object-oriented features of Smalltalk per se that have made a lasting impression but rather the then-innovative idea that every user would have a large graphical display and could interact with the system using a mouse and menus rather than by typing commands.

Guy Steele's *LAMBDA: The Ultimate Declarative* (1976a and b) was perhaps the first paper to demonstrate how object-oriented programming can be done in Lisp. As the title suggests, it was all done using `lambda`, in a similar way to our `define-class` example. Steele summarized the approach with the equation "Actors = Closures (mod Syntax)," referring to Carl Hewitt's "Actors" object-oriented formalism.

In 1979, the MIT Lisp Machine group developed the Flavors system based on this approach but offering considerable extensions (Cannon 1980, Weinreb 1980, Moon et al. 1983). "Flavor" was a popular jargon word for "type" or "kind" at MIT, so it was natural that it became the term for what we call classes.

The Flavor system was the first to support multiple inheritance. Other languages shunned multiple inheritance because it was too dynamic. With single inheritance, each instance variable and method could be assigned a unique offset number, and looking up a variable or method was therefore trivial. But with multiple inheritance, these computations had to be done at run time. The Lisp tradition enabled programmers to accept this dynamic computation, when other languages would not. Once it was accepted, the MIT group soon came to embrace it. They developed complex protocols for combining different flavors into new ones. The concept of *mix-ins* was developed by programmers who frequented Steve's Ice Cream parlor in nearby Davis Square. Steve's offered a list of ice cream flavors every day but also offered to create new flavors—dynamically—by mixing in various cookies, candies, or fruit, at the request of the individual customer. For example, Steve's did not have chocolate-chip ice cream on the menu, but you could always order vanilla ice cream with chocolate chips mixed in.<sup>3</sup>

This kind of "flavor hacking" appealed to the MIT Lisp Machine group, who

---

<sup>3</sup>Flavor fans will be happy to know that Steve's Ice Cream is now sold nationally in the United States. Alas, it is not possible to create flavors dynamically. Also, be warned that Steve's was bought out by his Teal Square rival, Joey's. The original Steve retired from the business for years, then came back with a new line of stores under his last name, Harrell.

adopted the metaphor for their object-oriented programming system. All flavors inherited from the top-most flavor in the hierarchy: `vanilla`. In the window system, for example, the flavor `basic-window` was defined to support the minimal functionality of all windows, and then new flavors of `window` were defined by combining mix-in flavors such as `scroll-bar-mixin`, `label-mixin`, and `border-mixin`. These mix-in flavors were used only to define other flavors. Just as you couldn't go into Steve's and order "crushed Heath bars, hold the ice cream," there was a mechanism to prohibit instantiation of mix-ins.

A complicated repertoire of *method combinations* was developed. The default method combination on Flavors was similar to CLOS: first do all the `:before` methods, then the most specific primary method, then the `:after` methods. But it was possible to combine methods in other ways as well. For example, consider the `inside-width` method, which returns the width in pixels of the usable portion of a window. A programmer could specify that the combined method for `inside-width` was to be computed by calling all applicable methods and summing them. Then an `inside-width` method for the `basic-window` flavor would be defined to return the width of the full window, and each mix-in would have a simple method to say how much of the width it consumed. For example, if borders are 8 pixels wide and scroll bars are 12 pixels wide, then the `inside-width` method for `border-mixin` returns -8 and `scroll-bar-mixin` returns -12. Then any window, no matter how many mix-ins it is composed of, automatically computes the proper inside width.

In 1981, Symbolics came out with a more efficient implementation of Flavors. Objects were no longer just closures. They were still funcallable, but there was additional hardware support that distinguished them from other functions. After a few years Symbolics abandoned the `(send object message)` syntax in favor of a new syntax based on generic functions. This system was known as New Flavors. It had a strong influence on the eventual CLOS design.

The other strong influence on CLOS was the CommonLoops system developed at Xerox PARC. (See Bobrow 1982, Bobrow et al. 1986, Stefik and Bobrow 1986.) CommonLoops continued the New Flavors trend away from message passing by introducing *multimethods*: methods that specialize on more than one argument.

As of summer 1991, CLOS itself is in a state of limbo. It was legitimized by its appearance in *Common Lisp the Language*, 2d edition, but it is not yet official, and an important part, the metaobject protocol, is not yet complete. A tutorial on CLOS is Keene 1989.

We have seen how easy it is to build an object-oriented system on top of Lisp, using `lambda` as the primary tool. An interesting alternative is to build Lisp on top of an object-oriented system. That is the approach taken in the Oaklisp system of Lang and Perlmutter (1988). Instead of defining methods using `lambda` as the primitive, Oaklisp has `add-method` as a primitive and defines `lambda` as a macro that adds a method to an anonymous, empty operation.

Of course, object-oriented systems are thriving outside the Lisp world. With the

success of UNIX-based workstations, C has become one of the most widely available programming languages. C is a fairly low-level language, so there have been several attempts to use it as a kind of portable assembly language. The most successful of these attempts is C++, a language developed by Bjarne Stroustrup of AT&T Bell Labs (Stroustrup 1986). C++ provides a number of extensions, including the ability to define classes. However, as an add-on to an existing language, it does not provide as many features as the other languages discussed here. Crucially, it does not provide garbage collection, nor does it support fully generic functions.

Eiffel (Meyer 1988) is an attempt to define an object-oriented system from the ground up rather than tacking it on to an existing language. Eiffel supports multiple inheritance and garbage collection and a limited amount of dynamic dispatching.

So-called modern languages like Ada and Modula support information-hiding through generic functions and classes, but they do not provide inheritance, and thus can not be classified as true object-oriented languages.

Despite these other languages, the Lisp-based object-oriented systems are the only ones since Smalltalk to introduce important new concepts: multiple inheritance and method combination from Flavors, and multimethods from CommonLoops.

## 13.12 Exercises

-  **Exercise 13.3 [m]** Implement `deposit` and `interest` methods for the `account` class using CLOS.
-  **Exercise 13.4 [m]** Implement the `password-account` class using CLOS. Can it be done as cleanly with inheritance as it was done with delegation? Or should you use delegation within CLOS?
-  **Exercise 13.5 [h]** Implement graph searching, search paths, and A\* searching as classes in CLOS.
-  **Exercise 13.6 [h]** Implement a priority queue to hold the states of a problem. Instead of a list, the `problem-states` will be a vector of lists, each initially null. Each new state will have a priority (determined by the generic function `priority`) which must be an integer between zero and the length of the vector, where zero indicates the highest priority. A new state with priority  $p$  is pushed onto element  $p$  of the vector, and the state to be explored next is the first state in the first nonempty position. As stated in the text, some of the previously defined methods made the unwarranted assumption that `problem-states` would always hold a list. Change these methods.

## CHAPTER 14

---

# Knowledge Representation and Reasoning

*Knowledge itself is power.*

—Francis Bacon (1561–1626)

*The power resides in the knowledge.*

—Edward Feigenbaum  
Stanford University Heuristic Programming Project

*Knowledge is Knowledge, and vice versa.*

—Tee shirt  
Stanford University Heuristic Programming Project

In the 1960s, much of AI concentrated on search techniques. In particular, a lot of work was concerned with *theorem proving*: stating a problem as a small set of axioms and searching for a proof of the problem. The implicit assumption was that the power resided in the inference mechanism—if we could just find the right search technique, then all our problems would be solved, and all our theorems would be proved.

Starting in the 1970s, this began to change. The theorem-proving approach failed to live up to its promise. AI workers slowly began to realize that they were not going to solve NP-hard problems by coming up with a clever inference algorithm. The general inferencing mechanisms that worked on toy examples just did not scale up when the problem size went into the thousands (or sometimes even into the dozens).

The *expert-system* approach offered an alternative. The key to solving hard problems was seen to be the acquisition of special-case rules to break the problem into easier problems. According to Feigenbaum, the lesson learned from expert systems like MYCIN (which we will see in chapter 16) is that the choice of inferencing mechanism is not as important as having the right knowledge. In this view it doesn't matter very much if MYCIN uses forward- or backward-chaining, or if it uses certainty factors, probabilities, or fuzzy set theory. What matters crucially is that we know *pseudomonas* is a gram-negative, rod-shaped organism that can infect patients with compromised immune systems. In other words, the key problem is acquiring and representing knowledge.

While the expert system approach had some successes, it also had failures, and researchers were interested in learning the limits of this new technology and understanding exactly how it works. Many found it troublesome that the meaning of the knowledge used in some systems was never clearly defined. For example, does the assertion (*color apple red*) mean that a particular apple is red, that all apples are red, or that some/most apples are red? The field of *knowledge representation* concentrated on providing clear semantics for such representations, as well as providing algorithms for manipulating the knowledge. Much of the emphasis was on finding a good trade-off between *expressiveness* and *efficiency*. An efficient language is one for which all queries (or at least the average query) can be answered quickly. If we want to guarantee that queries will be answered quickly, then we have to limit what can be expressed in the language.

In the late 1980s, a series of results shed doubt on the hopes of finding an efficient language with any reasonable degree of expressiveness at all. Using mathematical techniques based on worst-case analysis, it was shown that even seemingly trivial languages were *intractable*—in the worst case, it would take an exponential amount of time to answer a simple query.

Thus, in the 1990s the emphasis has shifted to *knowledge representation and reasoning*, a field that encompasses both the expressiveness and efficiency of languages but recognizes that the average case is more important than the worst case. No amount of knowledge can help solve an intractable problem in the worse case, but in practice the worst case rarely occurs.

## 14.1 A Taxonomy of Representation Languages

AI researchers have investigated hundreds of knowledge representation languages, trying to find languages that are convenient, expressive, and efficient. The languages can be classified into four groups, depending on what the basic unit of representation is. Here are the four categories, with some examples:

- *Logical Formulae* (Prolog)
- *Networks* (semantic nets, conceptual graphs)
- *Objects* (scripts, frames)
- *Procedures* (Lisp, production systems)

We have already dealt with *logic-based* languages like Prolog.

*Network-based* languages can be seen as a syntactic variation on logical languages. A link  $L$  between nodes  $A$  and  $B$  is just another way of expressing the logical relation  $L(A, B)$ . The difference is that network-based languages take their links more seriously: they are intended to be implemented directly by pointers in the computer, and inference is done by traversing these pointers. So placing a link  $L$  between  $A$  and  $B$  not only asserts that  $L(A, B)$  is true, but it also says something about how the knowledge base is to be searched.

*Object-oriented* languages can also be seen as syntactic variants of predicate calculus. Here is a statement in a typical slot-filler frame language:

```
(a person
  (name = Jan)
  (age = 32))
```

This is equivalent to the logical formula:

$$\exists p: \text{person}(p) \wedge \text{name}(p, \text{Jan}) \wedge \text{age}(p, 32)$$

The frame notation has the advantage of being easier to read, in some people's opinion. However, the frame notation is less expressive. There is no way to say that the person's name is either Jan or John, or that the person's age is not 34. In predicate calculus, of course, such statements can be easily made.

Finally, *procedural* languages are to be contrasted with representation languages: procedural languages compute answers without explicit representation of knowledge.

There are also hybrid representation languages that use different methods to encode different kinds of knowledge. The KL-ONE family of languages uses both logical formulae and objects arranged into a network, for example. Many frame

languages allow *procedural attachment*, a technique that uses arbitrary procedures to compute values for expressions that are inconvenient or impossible to express in the frame language itself.

## 14.2 Predicate Calculus and its Problems

So far, many of our representations have been based on predicate calculus, a notation with a distinguished position in AI: it serves as the universal standard by which other representations are defined and evaluated. The previous section gave an example expression from a frame language. The frame language may have many merits in terms of the ease of use of its syntax or the efficiency of its internal representation of data. However, to understand what expressions in the language mean, there must be a clear definition. More often than not, that definition is given in terms of predicate calculus.

A predicate calculus representation assumes a universe of individuals, with relations and functions on those individuals, and sentences formed by combining relations with the logical connectives and, or, and not. Philosophers and psychologists will argue the question of how appropriate predicate calculus is as a model of human thought, but one point stands clear: predicate calculus is sufficient to represent anything that can be represented in a digital computer. This is easy to show: assuming the computer's memory has  $n$  bits, and the equation  $b_i = 1$  means that bit  $i$  is on, then the entire state of the computer is represented by a conjunction such as:

$$(b_0 = 0) \wedge (b_1 = 0) \wedge (b_2 = 1) \wedge \cdots \wedge (b_n = 0)$$

Once we can represent a state of the computer, it becomes possible to represent any computer program in predicate calculus as a set of axioms that map one state onto another. Thus, predicate calculus is shown to be a *sufficient* language for representing anything that goes on inside a computer—it can be used as a tool for analyzing any program from the outside.

This does not prove that predicate calculus is an *appropriate* tool for all applications. There are good reasons why we may want to represent knowledge in a form that is quite different from predicate calculus, and manipulate the knowledge with procedures that are quite different from logical inference. But we should still be able to describe our system in terms of predicate calculus axioms, and prove theorems about it. To do any less is to be sloppy. For example, we may want to manipulate numbers inside the computer by using the arithmetic instructions that are built into the CPU rather than by manipulating predicate calculus axioms, but when we write a square-root routine, it had better satisfy the axiom:

$$\sqrt{x} = y \Rightarrow y \times y = x$$

Predicate calculus also serves another purpose: as a tool that can be used *by* a program rather than *on* a program. All programs need to manipulate data, and some programs will manipulate data that is considered to be in predicate calculus notation. It is this use that we will be concerned with.

Predicate calculus makes it easy to start writing down facts about a domain. But the most straightforward version of predicate calculus suffers from a number of serious limitations:

- *Decidability*—given a set of axioms and a goal, it may be that neither the goal nor its negation can be derived from the axioms.
- *Tractability*—even when a goal is provable, it may take too long to find the proof using the available inferencing mechanisms.
- *Uncertainty*—it can be inconvenient to deal with relations that are probable to a degree but not known to be definitely true or false.
- *Monotonicity*—in pure predicate calculus, once a theorem is proved, it is true forever. But we would like a way to derive tentative theorems that rely on assumptions, and be able to retract them when the assumptions prove false.
- *Consistency*—pure predicate calculus admits no contradictions. If by accident both  $P$  and  $\neg P$  are derived, then *any* theorem can be proved. In effect, a single contradiction corrupts the entire data base.
- *Omniscience*—it can be difficult to distinguish what is provable from what should be proved. This can lead to the unfounded assumption that an agent believes all the consequences of the facts it knows.
- *Expressiveness*—the first-order predicate calculus makes it awkward to talk about certain things, such as the relations and propositions of the language itself.

The view held predominantly today is that it is best to approach these problems with a dual attack that is both within and outside of predicate calculus. It is considered a good idea to invent new notations to address the problems—both for convenience and to facilitate special-purpose reasoners that are more efficient than a general-purpose theorem prover. However, it is also important to define scrupulously the meaning of the new notation in terms of familiar predicate-calculus notation. As Drew McDermott put it, “No notation without denotation!” (1978).

In this chapter we show how new notations (and their corresponding meanings) can be used to extend an existing representation and reasoning system. Prolog is chosen as the language to extend. This is not meant as an endorsement for Prolog as the ultimate knowledge representation language. Rather, it is meant solely to give us a clear and familiar foundation from which to build.

## 14.3 A Logical Language: Prolog

Prolog has been proposed as the answer to the problem of programming in logic. Why isn't it accepted as the universal representation language? Probably because Prolog is a compromise between a representation language and a programming language. Given two specifications that are logically equivalent, one can be an efficient Prolog program, while the other is not. Kowalski's famous equation "*algorithm = logic + control*" expresses the limits of logic alone: *logic = algorithm – control*. Many problems (especially in AI) have large or infinite search spaces, and if Prolog is not given some advice on how to search that space, it will not come up with the answer in any reasonable length of time.

Prolog's problems fall into three classes. First, in order to make the language efficient, its expressiveness was restricted. It is not possible to assert that a person's name is either Jan or John in Prolog (although it is possible to *ask* if the person's name is one of those). Similarly, it is not possible to assert that a fact is false; Prolog does not distinguish between false and unknown. Second, Prolog's inference mechanism is neither sound nor complete. Because it does not check for circular unification, it can give incorrect answers, and because it searches depth-first it can miss correct answers. Third, Prolog has no good way of adding control information to the underlying logic, making it inefficient on certain problems.

## 14.4 Problems with Prolog's Expressiveness

If Prolog is programming in logic, it is not the full predicate logic we are familiar with. The main problem is that Prolog can't express certain kinds of indefinite facts. It can represent definite facts: the capital of Rhode Island is Providence. It can represent conjunctions of facts: the capital of Rhode Island is Providence and the capital of California is Sacramento. But it can not represent disjunctions or negations: that the capital of California is *not* Los Angeles, or that the capital of New York is *either* New York City *or* Albany. We could try this:

```
(<- (not (capital LA CA))  
<- (or (capital Albany NY) (capital NYC NY)))
```

but note that these last two facts concern the relation *not* and *or*, not the relation *capital*. Thus, they will not be considered when we ask a query about *capital*. Fortunately, the assertion "Either NYC or Albany is the capital of NY" can be rephrased as two assertions: "Albany is the capital of NY if NYC is not" and "NYC is the capital of NY if Albany is not:"

```
(<- (capital Albany NY) (not (capital NYC NY)))
(<- (capital NYC NY) (not (capital Albany NY)))
```

Unfortunately, Prolog's `not` is different from logic's `not`. When Prolog answers "no" to a query, it means the query cannot be proven from the known facts. If everything is known, then the query must be false, but if there are facts that are not known, the query may in fact be true. This is hardly surprising; we can't expect a program to come up with answers using knowledge it doesn't have. But in this case, it causes problems. Given the previous two clauses and the query `(capital ?c NY)`, Prolog will go into an infinite loop. If we remove the first clause, Prolog would fail to prove that Albany is the capital, and hence conclude that NYC is. If we remove the second clause, the opposite conclusion would be drawn.

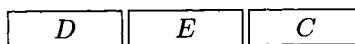
The problem is that Prolog equates "not proven" with "false." Prolog makes what is called the *closed world assumption*—it assumes that it knows everything that is true. The closed world assumption is reasonable for most programs, because the programmer does know all the relevant information. But for knowledge representation in general, we would like a system that does not make the closed world assumption and has three ways to answer a query: "yes," "no," or "unknown." In this example, we would not be able to conclude that the capital of NY is or is not NYC, hence we would not be able to conclude anything about Albany.

As another example, consider the clauses:

```
(<- (damned) (do))
(<- (damned) (not (do)))
```

With these rules, the query `(? (damned))` should logically be answered "yes." Furthermore, it should be possible to conclude `(damned)` without even investigating if `(do)` is provable or not. What Prolog does is first try to prove `(do)`. If this succeeds, then `(damned)` is proved. Either way, Prolog then tries again to prove `(do)`, and this time if the proof fails, then `(damned)` is proved. So Prolog is doing the same proof twice, when it is unnecessary to do the proof at all. Introducing negation wrecks havoc on the simple Prolog evaluation scheme. It is no longer sufficient to consider a single clause at a time. Rather, multiple clauses must be considered together if we want to derive all the right answers.

Robert Moore 1982 gives a good example of the power of disjunctive reasoning. His problem concerned three colored blocks, but we will update it to deal with three countries. Suppose that a certain Eastern European country, *E*, has just decided if it will remain under communist rule or become a democracy, but we do not know the outcome of the decision. *E* is situated between the democracy *D* and the communist country *C*:



The question is: Is there a communist country next to a democracy? Moore points out that the answer is “yes,” but discovering this requires reasoning by cases. If  $E$  is a democracy then it is next to  $C$  and the answer is yes. But if  $E$  is communist then it is next to  $D$  and the answer is still yes. Since those are the only two possibilities, the answer must be yes in any case. Logical reasoning gives us the right answer, but Prolog can not. We can describe the problem with the following seven assertions and one query, but Prolog can not deal with the or in the final assertion.

```
(<- (next-to D E))    (<- (next-to E D))
(<- (next-to E C))    (<- (next-to C E))
(<- (democracy D))   (<- (communist C))
(<- (or (democracy E) (communist E)))

(?- (next-to ?A ?B) (democracy ?A) (communist ?B))
```

We have seen that Prolog is not very good at representing disjunctions and negations. It also has difficulty representing existentials. Consider the following statement in English, logic, and Prolog:

Jan likes everyone.  
 $\forall x \text{ person}(x) \Rightarrow \text{likes}(\text{Jan}, x)$   
 $(<- (\text{likes Jan } ?x) (\text{person } ?x))$

The Prolog translation is faithful. But there is no good translation for “Jan likes someone.” The closest we can get is:

Jan likes someone.  
 $\exists x \text{ person}(x) \Rightarrow \text{likes}(\text{Jan}, x)$   
 $(<- (\text{likes Jan p1}))$   
 $(<- (\text{person p1}))$

Here we have invented a new symbol,  $p1$ , to represent the unknown person that Jan likes, and have asserted that  $p1$  is a person. Notice that  $p1$  is a constant, not a variable. This use of a constant to represent a specific but unknown entity is called a *Skolem constant*, after the logician Thoralf Skolem (1887–1963). The intent is that  $p1$  may be equal to some other person that we know about. If we find out that Adrian is the person Jan likes, then in logic we can just add the assertion  $p1 = \text{Adrian}$ . But that does not work in Prolog, because Prolog implicitly uses the *unique name assumption*—all atoms represent distinct individuals.

A Skolem constant is really just a special case of a *Skolem function*—an unknown entity that depends on one or more variables. For example, to represent “Everyone likes someone” we could use:

Everyone likes someone.

$$\forall y \exists x \text{person}(x) \Rightarrow \text{likes}(y, x)$$

$$(\leftarrow (\text{likes} ?y (\text{p2} ?y)))$$

$$(\leftarrow (\text{person} (\text{p2} ?y)))$$

Here p2 is a Skolem function that depends on the variable ?y. In other words, everyone likes some person, but not necessarily the same person.

## 14.5 Problems with Predicate Calculus's Expressiveness

In the previous section we saw that Prolog has traded some expressiveness for efficiency. This section explores the limits of predicate calculus's expressiveness.

Suppose we want to assert that lions, tigers, and bears are kinds of animals. In predicate calculus or in Prolog we could write an implication for each case:

```
(\leftarrow (\text{animal} ?x) (\text{lion} ?x))
(\leftarrow (\text{animal} ?x) (\text{tiger} ?x))
(\leftarrow (\text{animal} ?x) (\text{bear} ?x))
```

These implications allow us to prove that any known lion, tiger, or bear is in fact an animal. However, they do not allow us to answer the question "What kinds of animals are there?" It is not hard to imagine extending Prolog so that the query

```
(?- (\leftarrow (\text{animal} ?x) ?proposition))
```

would be legal. However, this happens not to be valid Prolog, and it is not even valid first-order predicate calculus (or FOPC). In FOPC the variables must range over constants in the language, not over relations or propositions. Higher-order predicate calculus removes this limitation, but it has a more complicated proof theory.

It is not even clear what the values of ?proposition should be in the query above. Surely (`lion ?x`) would be a valid answer, but so would (`animal ?x`), (or (`tiger ?x`) (`bear ?x`)), and an infinite number of other propositions. Perhaps we should have two types of queries, one that asks about "kinds," and another that asks about propositions.

There are other questions that we might want to ask about relations. Just as it is useful to declare the types of parameters to a Lisp function, it can be useful to declare the types of the parameters of a relation, and later query those types. For example, we might say that the `likes` relation holds between a person and an object.

In general, a sentence in the predicate calculus that uses a relation or sentence as a term is called a higher-order sentence. There are some quite subtle problems that

come into play when we start to allow higher-order expressions. Allowing sentences in the calculus to talk about the truth of other sentences can lead to a paradox: is the sentence “This sentence is false” true or false?

Predicate calculus is defined in terms of a universe of individuals and their properties and relations. Thus it is well suited for a model of the world that picks out individuals and categorizes them—a person here, a building there, a sidewalk between them. But how well does predicate calculus fare in a world of continuous substances? Consider a body of water consisting of an indefinite number of subconstituents that are all water, with some of the water evaporating into the air and rising to form clouds. It is not at all obvious how to define the individuals here. However, Patrick Hayes has shown that when the proper choices are made, predicate calculus can describe this kind of situation quite well. The details are in Hayes 1985.

The need to define categories is a more difficult problem. Predicate calculus works very well for crisp, mathematical categories:  $x$  is a triangle if and only if  $x$  is a polygon with three sides. Unfortunately, most categories that humans deal with in everyday life are not defined so rigorously. The category *friend* refers to someone you have mostly positive feelings for, whom you can usually trust, and so on. This “definition” is not a set of necessary and sufficient conditions but rather is an open-ended list of ill-defined qualities that are highly correlated with the category *friend*. We have a prototype for what an ideal friend should be, but no clear-cut boundaries that separate *friend* from, say, *acquaintance*. Furthermore, the boundaries seem to vary from one situation to another: a person you describe as a good friend in your work place might be only an acquaintance in the context of your home life.

There are versions of predicate calculus that admit quantifiers like “most” in addition to “for all” and “there exists,” and there have been attempts to define prototypes and measure distances from them. However, there is no consensus on the way to approach this problem.

## 14.6 Problems with Completeness

Because Prolog searches depth-first, it can get caught in one branch of the search space and never examine the other branches. This problem can show up, for example, in trying to define a commutative relation, like `sibling`:

```
(<- (sibling lee kim))  
(<- (sibling ?x ?y) (sibling ?y ?x))
```

With these clauses, we expect to be able to conclude that Lee is Kim’s sibling, and Kim is Lee’s. Let’s see what happens:

```
> (?- (sibling ?x ?y))
?X = LEE
?Y = KIM;
?X = KIM
?Y = LEE;
?X = LEE
?Y = KIM;
?X = KIM
?Y = LEE.
No.
```

We get the expected conclusions, but they are deduced repeatedly, because the commutative clause for siblings is applied over and over again. This is annoying, but not critical. Far worse is when we ask (?- (sibling fred ?x)). This query loops forever. Happily, this particular type of example has an easy fix: just introduce two predicates, one for data-base level facts, and one at the level of axioms and queries:

```
(<- (sibling-fact lee kim))
(<- (sibling ?x ?y) (sibling-fact ?x ?y))
(<- (sibling ?x ?y) (sibling-fact ?y ?x))
```

Another fix would be to change the interpreter to fail when a repeated goal was detected. This was the approach taken in GPS. However, even if we eliminated repeated goals, Prolog can still get stuck in one branch of a depth-first search. Consider the example:

```
(<- (natural 0))
(<- (natural (1+ ?n)) (natural ?n))
```

These rules define the natural numbers (the non-negative integers). We can use the rules either to confirm queries like (natural (1+ (1+ (1+ 0)))) or to generate the natural numbers, as in the query (natural ?n). So far, everything is fine. But suppose we wanted to define all the integers. One approach would be this:

```
(<- (integer 0))
(<- (integer ?n) (integer (1+ ?n)))
(<- (integer (1+ ?n)) (integer ?n))
```

These rules say that 0 is an integer, and any  $n$  is an integer if  $n + 1$  is, and  $n + 1$  is if  $n$  is. While these rules are correct in a logical sense, they don't work as a Prolog program. Asking (integer  $x$ ) will result in an endless series of ever-increasing queries: (integer (1+  $x$ )), (integer (1+ (1+  $x$ ))), and so on. Each goal is different, so no check can stop the recursion.

The occurs check may or may not introduce problems into Prolog, depending on your interpretation of infinite trees. Most Prolog systems do not do the occurs check. The reasoning is that unifying a variable with some value is the Prolog equivalent of assigning a value to a variable, and programmers expect such a basic operation to be fast. With the occurs check turned off, it will in fact be fast. With checking on, it takes time proportional to the size of the value, which is deemed unacceptable.

With occurs checking off, the programmer gets the benefit of fast unification but can run into problems with circular structures. Consider the following clauses:

```
(<- (parent ?x (mother-of ?x)))  
(<- (parent ?x (father-of ?x)))
```

These clauses say that, for any person, the mother of that person and the father of that person are parents of that person. Now let us ask if there is a person who is his or her own parent:

```
> (? (parent ?y ?y))  
?Y = [Abort]
```

The system has found an answer, where  $?y = (\text{mother-of } ?y)$ . The answer can't be printed, though, because `deref` (or `subst-bindings` in the interpreter) goes into an infinite loop trying to figure out what  $?y$  is. Without the printing, there would be no infinite loop:

```
(<- (self-parent) (parent ?y ?y))  
  
> (? (self-parent))  
Yes;  
Yes;  
No.
```

The self-parent query succeeds twice, once with the mother clause and once with the father clause. Has Prolog done the right thing here? It depends on your interpretation of infinite circular trees. If you accept them as valid objects, then the answer is consistent. If you don't, then leaving out the occurs check makes Prolog *unsound*: it can come up with incorrect answers.

The same problem comes up if we ask if there are any sets that include themselves as members. The query `(member ?set ?set)` will succeed, but we will not be able to print the value of `?set`.

## 14.7 Problems with Efficiency: Indexing

Our Prolog compiler is designed to handle “programlike” predicates—predicates with a small number of rules, perhaps with complex bodies. The compiler does much worse on “tablelike” predicates—predicates with a large number of simple facts. Consider the predicate `pb`, which encodes phone-book facts in the form:

```
(pb (name Jan Doe) (num 415 555 1212))
```

Suppose we have a few thousand entries of this kind. A typical query for this data base would be:

```
(pb (name Jan Doe) ?num)
```

It would be inefficient to search through the facts linearly, matching each one against the query. It would also be inefficient to recompile the whole `pb/2` predicate every time a new entry is added. But that is just what our compiler does.

The solutions to the three problems—expressiveness, completeness, and indexing—will be considered in reverse order, so that the most difficult one, expressiveness, will come last.

## 14.8 A Solution to the Indexing Problem

A better solution to the phone-book problem is to index each phone-book entry in some kind of table that makes it easy to add, delete, and retrieve entries. That is what we will do in this section. We will develop an extension of the trie or discrimination tree data structure built in section 10.5 (page 344).

Making a discrimination tree for Prolog facts is complicated by the presence of variables in both the facts and the query. Either facts with variables in them will have to be indexed in several places, or queries with variables will have to look in several places, or both. We also have to decide if the discrimination tree itself will handle variable binding, or if it will just return candidate matches which are then checked by some other process. It is not clear what to store in the discrimination tree: copies of the fact, functions that can be passed continuations, or something else. More design choices will come up as we proceed.

It is difficult to make design choices when we don’t know exactly how the system will be used. We don’t know what typical facts will look like, nor typical queries. Therefore, we will design a fairly abstract tool, forgetting for the moment that it will be used to index Prolog facts.

We will address the problem of a discrimination tree where both the keys and queries are predicate structures with wild cards. A wild card is a variable, but with the understanding that there is no variable binding; each instance of a variable can match anything. A predicate structure is a list whose first element is a nonvariable symbol. The discrimination tree supports three operations:

- index—add a key/value pair to the tree
- fetch—find all values that potentially match a given key
- unindex—remove all key/value pairs that match a given key

To appreciate the problems, we need an example. Suppose we have the following six keys to index. For simplicity, the value of each key will be the key itself:

```

1 (p a b)
2 (p a c)
3 (p a ?x)
4 (p b c)
5 (p b (f c))
6 (p a (f . ?x))

```

Now assume the query  $(p \ ?y \ c)$ . This should match keys 2, 3, and 4. How could we efficiently arrive at this set? One idea is to list the key/value pairs under every atom that they contain. Thus, all six would be listed under the atom  $p$ , while 2, 4, and 5 would be listed under the atom  $c$ . A unification check could eliminate 5, but we still would be missing 3. Key 3 (and every key with a variable in it) could potentially contain the atom  $c$ . So to get the right answers under this approach, we will need to index every key that contains a variable under every atom—not an appealing situation.

An alternative is to create indices based on both atoms and their position. So now we would be retrieving all the keys that have a  $c$  in the second argument position: 2 and 4, plus the keys that have a variable as the second argument: 3. This approach seems to work much better, at least for the example shown. To create the index, we essentially superimpose the list structure of all the keys on top of each other, to arrive at one big discrimination tree. At each position in the tree, we create an index of the keys that have either an atom or a variable at that position. Figure 14.1 shows the discrimination tree for the six keys.

Consider the query  $(p \ ?y \ c)$ . Either the  $p$  or the  $c$  could be used as an index. The  $p$  in the predicate position retrieves all six keys. But the  $c$  in the second argument position retrieves only three keys: 2 and 4, which are indexed under  $c$  itself, and 3, which is indexed under the variable in that position.

Now consider the query  $(p \ ?y \ (f \ ?z))$ . Again, the  $p$  serves as an index to all six keys. The  $f$  serves as an index to only three keys: the 5 and 6, which are indexed

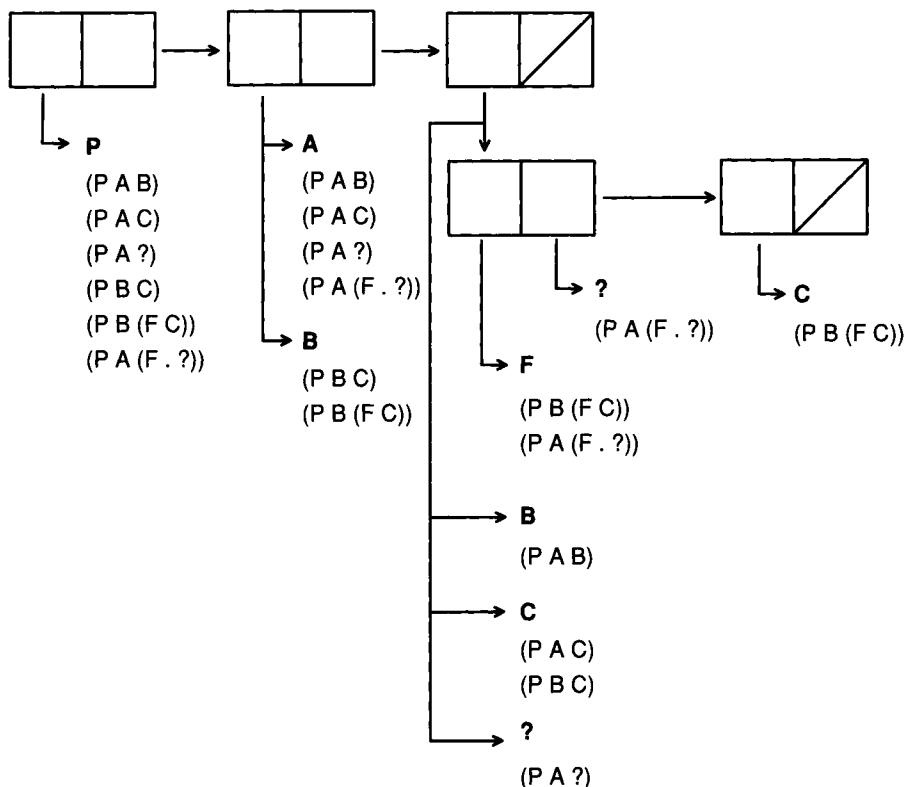


Figure 14.1: Discrimination Tree with Six Keys

directly under *f* in that position, and 3, which is indexed under the variable in a position along the path that lead to *f*. In general, all the keys indexed under variables along the path must be considered.

The retrieval mechanism can overretrieve. Given the query  $(p\ a\ (f\ ?x))$ , the atom *p* will again retrieve all six keys, the atom *a* retrieves 1, 2, 3, and 6, and *f* again retrieves 5, 6, and 3. So *f* retrieves the shortest list, and hence it will be used to determine the final result. But key 5 is  $(p\ b\ (f\ c))$ , which does not match the query  $(p\ a\ (f\ ?x))$ .

We could eliminate this problem by intersecting all the lists instead of just taking the shortest list. It is perhaps feasible to do the intersection using bit vectors, but probably too slow and wasteful of space to do it using lists. Even if we did intersect keys, we would still overretrieve, for two reasons. First, we don't use *nil* as an index, so we are ignoring the difference between  $(f\ ?x)$  and  $(f\ .\ ?x)$ . Second, we are using wild-card semantics, so the query  $(p\ ?x\ ?x)$  would retrieve all six keys, when

it should only retrieve three. Because of these problems, we make a design choice: we will first build a data base retrieval function that retrieves potential matches, and later worry about the unification process that will eliminate mismatches.

We are ready for a more complete specification of the indexing strategy:

- The value will be indexed under each non-nil nonvariable atom in the key, with a separate index for each position. For example, given the preceding data base, the atom *a* in the first argument position would index values 1, 2, 3, and 6, while the atom *b* in the second argument position would index value 4 and 5. The atom *p* in the predicate position would index all six values.
- In addition, we will maintain a separate index for variables at each position. For example, value 3 would be stored under the index “variable in second argument position.”
- “Position” does not refer solely to the linear position in the top-level list. For example, value 5 would be indexed under atom *f* in the *caaddr* position.
- It follows that a key with *n* atoms will be indexed in *n* different ways.

For retrieval, the strategy is:

- For each non-nil nonvariable atom in the retrieval key, generate a list of possible matches. Choose the shortest such list.
- Each list of possible matches will have to be augmented with the values indexed under a variable at every position “above.” For example, *f* in the *caaddr* position retrieves value 5, but it also must retrieve value 3, because the third key has a variable in the *caddr* position, and *caddr* is “above” *caaddr*.
- The discrimination tree may return values that are not valid matches. The purpose of the discrimination tree is to reduce the number of values we will have to unify against, not to determine the exact set of matches.

It is important that the retrieval function execute quickly. If it is slow, we might just as well match against every key in the table linearly. Therefore, we will take care to implement each part efficiently. Note that we will have to compare the length of lists to choose the shortest possibility. Of course, it is trivial to compare lengths using *length*, but *length* requires traversing the whole list. We can do better if we store the length of the list explicitly. A list with its length will be called an *nlist*. It will be implemented as a cons cell containing the number of elements and a list of the elements themselves. An alternative would be to use extensible vectors with fill pointers.

```
;; An nlist is implemented as a (count . elements) pair:
(defun make-empty-nlist ()
  "Create a new, empty nlist."
  (cons 0 nil))

(defun nlist-n (x) "The number of elements in an nlist." (car x))
(defun nlist-list (x) "The elements in an nlist." (cdr x))

(defun nlist-push (item nlist)
  "Add a new element to an nlist."
  (incf (car nlist))
  (push item (cdr nlist))
  nlist)
```

Now we need a place to store these nlists. We will build the data base out of discrimination tree nodes called dtree nodes. Each dtree node has a field to hold the variable index, the atom indices, and pointers to two subnodes, one for the first and one for the rest. We implement dtrees as vectors for efficiency, and because we will never need a dtree-p predicate.

```
(defstruct (dtree (:type vector))
  (first nil) (rest nil) (atoms nil) (var (make-empty-nlist)))
```

A separate dtree will be stored for each predicate. Since the predicates must be symbols, it is possible to store the dtrees on the predicate's property list. In most implementations, this will be faster than alternatives such as hash tables.

```
(let ((predicates nil))

  (defun get-dtree (predicate)
    "Fetch (or make) the dtree for this predicate."
    (cond ((get predicate 'dtree))
          (t (push predicate predicates)
              (setf (get predicate 'dtree) (make-dtree)))))

  (defun clear-dtrees ()
    "Remove all the dtrees for all the predicates."
    (dolist (predicate predicates)
      (setf (get predicate 'dtree) nil)))
    (setf predicates nil)))
```

The function index takes a relation as key and stores it in the dtree for the predicate of the relation. It calls dtree-index to do all the work of storing a value under the proper indices for the key in the proper dtree node.

The atom indices are stored in an association list. Property lists would not work, because they are searched using eq and atoms can be numbers, which are not

necessarily eq. Association lists are searched using eql by default. An alternative would be to use hash tables for the index, or even to use a scheme that starts with association lists and switches to a hash table when the number of entries gets large. I use lookup to look up the value of a key in a property list. This function, and its setf method, are defined on page 896.

```
(defun index (key)
  "Store key in a dtree node. Key must be (predicate . args);
  it is stored in the predicate's dtree."
  (dtree-index key key (get-dtree (predicate key)))))

(defun dtree-index (key value dtree)
  "Index value under all atoms of key in dtree."
  (cond
    ((consp key) ; index on both first and rest
     (dtree-index (first key) value
                  (or (dtree-first dtree)
                      (setf (dtree-first dtree) (make-dtree))))
     (dtree-index (rest key) value
                  (or (dtree-rest dtree)
                      (setf (dtree-rest dtree) (make-dtree)))))
    ((null key)) ; don't index on nil
    ((variable-p key)) ; index a variable
    (nlist-push value (dtree-var dtree)))
    (t ; Make sure there is an nlist for this atom, and add to it
     (nlist-push value (lookup-atom key dtree)))))

(defun lookup-atom (atom dtree)
  "Return (or create) the nlist for this atom in dtree."
  (or (lookup atom (dtree-atoms dtree))
      (let ((new (make-empty-nlist)))
        (push (cons atom new) (dtree-atoms dtree))
        new)))
```

Now we define a function to test the indexing routine. Compare the output with figure 14.1.

```
(defun test-index ()
  (let ((props '((p a b) (p a c) (p a ?x) (p b c)
                (p b (f c)) (p a (f . ?x))))))
    (clear-dtrees)
    (mapc #'index props)
    (write (list props (get-dtree 'p))
          :circle t :array t :pretty t)
    (values)))
```

```
> (test-index)
((#1=(P A B)
 #2=(P A C)
 #3=(P A ?X)
 #4=(P B C)
 #5=(P B (F C))
 #6=(P A (F . ?X)))
 #(#(NIL NIL (P (6 #6# #5# #4# #3# #2# #1#)) (0))
 #(#(NIL NIL (B (2 #5# #4#) A (4 #6# #3# #2# #1#)) (0))
 #(#(#(NIL NIL (F (2 #6# #5#)) (0))
 #(#(NIL NIL (C (1 #5#)) (0))
 #(NIL NIL NIL (0)) NIL (1 #6#))
 (C (2 #4# #2#) B (1 #1#))
 (1 #3#))
 #(NIL NIL NIL (0))
 NIL (0))
 NIL (0)))
```

The next step is to fetch matches from the dtree data base. The function `fetch` takes a query, which must be a valid relation, as its argument, and returns a list of possible matches. It calls `dtree-fetch` to do the work:

```
(defun fetch (query)
  "Return a list of buckets potentially matching the query,
  which must be a relation of form (predicate . args)."
  (dtree-fetch query (get-dtree (predicate query))
    nil 0 nil most-positive-fixnum))
```

`dtree-fetch` must be passed the query and the dtree, of course, but it is also passed four additional arguments. First, we have to accumulate matches indexed under variables as we are searching through the dtree. So two arguments are used to pass the actual matches and a count of their total number. Second, we want `dtree-fetch` to return the shortest possible index, so we pass it the shortest answer found so far, and the size of the shortest answer. That way, as it is making its way down the tree, accumulating values indexed under variables, it can be continually comparing the size of the evolving answer with the best answer found so far.

We could use `nlists` to pass around count/values pairs, but `nlists` only support a push operation, where one new item is added. We need to append together lists of values coming from the variable indices with values indexed under an atom. Append is expensive, so instead we make a list-of-lists and keep the count in a separate variable. When we are done, `dtree-fetch` and hence `fetch` does a multiple-value return, yielding the list-of-lists and the total count.

There are four cases to consider in dtree-fetch. If the dtree is null or the query pattern is either null or a variable, then nothing will be indexed, so we should just return the best answer found so far. Otherwise, we bind var-n and var-list to the count and list-of-lists of variable matches found so far, including at the current node. If the count var-n is greater than the best count so far, then there is no sense continuing, and we return the best answer found. Otherwise we look at the query pattern. If it is an atom, we use dtree-atom-fetch to return either the current index (along with the accumulated variable index) or the accumulated best answer, whichever is shorter. If the query is a cons, then we use dtree-fetch on the first part of the cons, yielding a new best answer, which is passed along to the call of dtree-fetch on the rest of the cons.

```
(defun dtree-fetch (pat dtree var-list-in var-n-in best-list best-n)
  "Return two values: a list-of-lists of possible matches to pat,
  and the number of elements in the list-of-lists."
  (if (or (null dtree) (null pat) (variable-p pat))
      (values best-list best-n)
      (let* ((var-nlist (dtree-var dtree))
             (var-n (+ var-n-in (nlist-n var-nlist)))
             (var-list (if (null (nlist-list var-nlist))
                         var-list-in
                         (cons (nlist-list var-nlist)
                               var-list-in))))
        (cond
          ((>= var-n best-n) (values best-list best-n))
          ((atom pat) (dtree-atom-fetch pat dtree var-list var-n
                                         best-list best-n))
          (t (multiple-value-bind (list1 n1)
                  (dtree-fetch (first pat) (dtree-first dtree)
                               var-list var-n best-list best-n)
                  (dtree-fetch (rest pat) (dtree-rest dtree)
                               var-list var-n list1 n1))))))

  (defun dtree-atom-fetch (atom dtree var-list var-n best-list best-n)
    "Return the answers indexed at this atom (along with the vars),
    or return the previous best answer, if it is better."
    (let ((atom-nlist (lookup atom (dtree-atoms dtree))))
      (cond
        ((or (null atom-nlist) (null (nlist-list atom-nlist)))
         (values var-list var-n))
        ((and atom-nlist (< (incf var-n (nlist-n atom-nlist)) best-n))
         (values (cons (nlist-list atom-nlist) var-list) var-n))
        (t (values best-list best-n))))
```

Here we see a call to fetch on the data base created by test-index. It returns two values: a list-of-lists of facts, and the total number of facts, three.

```
> (fetch '(p ? c))
(((P B C) (P A C))
 ((P A ?X)))
3
```

Now let's stop and see what we have accomplished. The functions `fetch` and `dtree-fetch` fulfill their contract of returning potential matches. However, we still need to integrate the `dtree` facility with Prolog. We need to go through the potential matches and determine which candidates are actual matches. For simplicity we will use the version of `unify` with binding lists defined in section 11.2. (It is also possible to construct a more efficient version that uses the compiler and the destructive function `unify!.`)

The function `mapc-retrieve` calls `fetch` to get a list-of-lists of potential matches and then calls `unify` to see if the match is a true one. If the match is true, it calls the supplied function with the binding list that represents the unification as the argument. `mapc-retrieve` is proclaimed `inline` so that functions passed to it can also be compiled in place.

```
(proclaim '(inline mapc-retrieve))

(defun mapc-retrieve (fn query)
  "For every fact that matches the query,
  apply the function to the binding list."
  (dolist (bucket (fetch query))
    (dolist (answer bucket)
      (let ((bindings (unify query answer)))
        (unless (eq bindings fail)
          (funcall fn bindings))))))
```

There are many ways to use this retriever. The function `retrieve` returns a list of the matching binding lists, and `retrieve-matches` substitutes each binding list into the original query so that the result is a list of expressions that unify with the query.

```
(defun retrieve (query)
  "Find all facts that match query.  Return a list of bindings."
  (let ((answers nil))
    (mapc-retrieve #'(lambda (bindings) (push bindings answers))
                  query)
    answers))

(defun retrieve-matches (query)
  "Find all facts that match query.
  Return a list of expressions that match the query."
  (mapcar #'(lambda (bindings) (subst-bindings bindings query))
          (retrieve query)))
```

There is one further complication to consider. Recall that in our original Prolog interpreter, the function `prove` had to rename the variables in each clause as it retrieved it from the data base. This was to insure that there was no conflict between the variables in the query and the variables in the clause. We could do that in `retrieve`. However, if we assume that the expressions indexed in discrimination trees are tablelike rather than rulelike and thus are not recursive, then we can get away with renaming the variables only once, when they are entered into the data base. This is done by changing `index`:

```
(defun index (key)
  "Store key in a dtree node. Key must be (predicate . args);
  it is stored in the predicate's dtree."
  (dtree-index key (rename-variables key)      ; store unique vars
                (get-dtree (predicate key))))
```

With the new `index` in place, and after calling `test-index` to rebuild the data base, we are now ready to test the retrieval mechanism:

```
> (fetch '(p ?x c))
(((P B C) (P A C))
 ((P A ?X3408)))
3

> (retrieve '(p ?x c))
((?X3408 . C) (?X . A))
((?X . A))
((?X . B))

> (retrieve-matches '(p ?x c))
((P A C) (P A C) (P B C))

> (retrieve-matches '(p ?x (?fn c)))
((P A (?FN C)) (P A (F C)) (P B (F C)))
```

Actually, it is better to use `mapc-retrieve` when possible, since it doesn't cons up answers the way `retrieve` and `retrieve-matches` do. The macro `query-bind` is provided as a nice interface to `mapc-retrieve`. The macro takes as arguments a list of variables to bind, a query, and one or more forms to apply to each retrieved answer. Within this list of forms, the variables will be bound to the values that satisfy the query. The syntax was chosen to be the same as `multiple-value-bind`. Here we see a typical use of `query-bind`, its result, and its macro-expansion:

```

> (query-bind (?x ?fn) '(p ?x (?fn c))
  (format t "~&P holds between ~a and ~a of c." ?x ?fn)) =>
P holds between B and F of c.
P holds between A and F of c.
P holds between A and ?FN of c.
NIL

≡ (mapc-retrieve
 #'(lambda (#:bindings6369)
 (let ((?x (subst-bindings #:bindings6369 '?x))
      (?fn (subst-bindings #:bindings6369 '?fn)))
   (format t "~&P holds between ~a and ~a of c." ?x ?fn)))
 '(p ?x (?fn c)))

```

Here is the implementation:

```

(defmacro query-bind (variables query &body body)
  "Execute the body for each match to the query.
  Within the body, bind each variable."
  (let* ((bindings (gensym "BINDINGS"))
         (vars-and-vals
          (mapcar
           #'(lambda (var)
               (list var '(subst-bindings ,bindings ',var)))
           variables)))
    '(mapc-retrieve
      #'(lambda (,bindings)
          (let ,vars-and-vals
              ,@body))
      ,query)))

```

## 14.9 A Solution to the Completeness Problem

We saw in chapter 6 that iterative deepening is an efficient way to cover a search space without falling into an infinite loop. Iterative deepening can also be used to guide the search in Prolog. It will insure that all valid answers are found eventually, but it won't turn an infinite search space into a finite one.

In the interpreter, iterative deepening is implemented by passing an extra argument to prove and prove-all to indicate the depth remaining to be searched. When that argument is zero, the search is cut off, and the proof fails. On the next iteration the bounds will be increased and the proof may succeed. If the search is never cut off by a depth bound, then there is no reason to go on to the next iteration, because all

proofs have already been found. The special variable \*search-cut-off\* keeps track of this.

```
(defvar *search-cut-off* nil "Has the search been stopped?")

(defun prove-all (goals bindings depth)
  "Find a solution to the conjunction of goals."
  ;; This version just passes the depth on to PROVE.
  (cond ((eq bindings fail) fail)
        ((null goals) bindings)
        (t (prove (first goals) bindings (rest goals) depth)))))

(defun prove (goal bindings other-goals depth)
  "Return a list of possible solutions to goal."
  ;; Check if the depth bound has been exceeded
  (if (= depth 0) ;***
      (progn (setf *search-cut-off* t) ;***
             fail) ;***
      (let ((clauses (get-clauses (predicate goal))))
        (if (listp clauses)
            (some
              #'(lambda (clause)
                  (let ((new-clause (rename-variables clause)))
                    (prove-all
                      (append (clause-body new-clause) other-goals)
                      (unify goal (clause-head new-clause) bindings)
                      (- depth 1))) ;***
                      clauses)
                  ;; The predicate's "clauses" can be an atom:
                  ;; a primitive function to call
                  (funcall clauses (rest goal) bindings
                            other-goals depth)))) ;***
```

`prove` and `prove-all` now implement search cutoff, but we need something to control the iterative deepening of the search. First we define parameters to control the iteration: one for the initial depth, one for the maximum depth, and one for the increment between iterations. Setting the initial and increment values to one will make the results come out in strict breadth-first order, but will duplicate more effort than a slightly larger value.

```
(defparameter *depth-start* 5
  "The depth of the first round of iterative search.")
(defparameter *depth-incr* 5
  "Increase each iteration of the search by this amount.")
(defparameter *depth-max* most-positive-fixnum
  "The deepest we will ever search.")
```

A new version of top-level-prove will be used to control the iteration. It calls prove-all for all depths from the starting depth to the maximum depth, increasing by the increment. However, it only proceeds to the next iteration if the search was cut off at some point in the previous iteration.

```
(defun top-level-prove (goals)
  (let ((all-goals
         `(',@goals (show-prolog-vars ,(variables-in goals))))))
    (loop for depth from *depth-start* to *depth-max* by *depth-incr*
          while (let ((*search-cut-off* nil))
                  (prove-all all-goals no-bindings depth
                             *search-cut-off*)))
          (format t "~&No.")
          (values)))
```

There is one final complication. When we increase the depth of search, we may find some new proofs, but we will also find all the old proofs that were found on the previous iteration. We can modify show-prolog-vars to only print proofs that are found with a depth less than the increment—that is, those that were not found on the previous iteration.

```
(defun show-prolog-vars (vars bindings other-goals depth)
  "Print each variable with its binding.
  Then ask the user if more solutions are desired."
  (if (> depth *depth-incr*)
      fail
      (progn
        (if (null vars)
            (format t "~&Yes")
            (dolist (var vars)
              (format t "~&~a = ~a" var
                     (subst-bindings bindings var))))
        (if (continue-p)
            fail
            (prove-all other-goals bindings depth))))))
```

To test that this works, try setting \*depth-max\* to 5 and running the following assertions and query. The infinite loop is avoided, and the first four solutions are found.

```

(<- (natural 0))
(<- (natural (1+ ?n)) (natural ?n))

> (?- (natural ?n))
?N = 0;
?N = (1+ 0);
?N = (1+ (1+ 0));
?N = (1+ (1+ (1+ 0)));
?N = (1+ (1+ (1+ (1+ 0))));
No.

```

## 14.10 Solutions to the Expressiveness Problems

In this section we present solutions to three of the limitations described above:

- Treatment of (limited) higher-order predication.
- Introduction of a frame-based syntax.
- Support for possible worlds, negation, and disjunction.

We also introduce a way to attach functions to predicates to do forward-chaining and error detection, and we discuss ways to extend unification to handle Skolem constants and other problems.

### *Higher-Order Predication*

First we will tackle the problem of answering questions like “What kinds of animals are there?” Paradoxically, the key to allowing more expressiveness in this case is to invent a new, more limited language and insist that all assertions and queries are made in that language. That way, queries that would have been higher-order in the original language become first-order in the restricted language.

The language admits three types of objects: *categories*, *relations*, and *individuals*. A category corresponds to a one-place predicate, a relation to a two-place predicate, and an individual to constant, or zero-place predicate. Statements in the language must have one of five primitive operators: `sub`, `rel`, `ind`, `val`, and `and`. They have the following form:

```

(sub subcategory supercategory)
(rel relation domain-category range-category)
(ind individual category)
(val relation individual value)
(and assertion ...)

```

The following table gives some examples, along with English translations:

(sub dog animal)	Dog is a kind of animal.
(rel birthday animal date)	The birthday relation holds between each animal and some date.
(ind fido dog)	The individual Fido is categorized as a dog.
(val birthday fido july-1)	The birthday of Fido is July-1.
(and A B)	Both A and B are true.

For those who feel more comfortable with predicate calculus, the following table gives the formal definition of each primitive. The most complicated definition is for *rel*. The form (*rel R A B*) means that every *R* holds between an individual of *A* and an individual of *B*, and furthermore that every individual of *A* participates in at least one *R* relation.

(sub A B)	$\forall x : A(x) \supset B(x)$
(rel R A B)	$\forall x, y : R(x, y) \supset A(x) \wedge B(y)$ $\wedge \forall x A(x) \supset \exists y : R(x, y)$
(ind I C)	$C(I)$
(val R I V)	$R(I, V)$
(and P Q...)	$P \wedge Q \dots$

Queries in the language, not surprisingly, have the same form as assertions, except that they may contain variables as well as constants. Thus, to find out what kinds of animals there are, use the query (sub ?kind animal). To find out what individual animals there are, use the query (ind ?x animal). To find out what individual animals of what kinds there are, use:

```
(and (sub ?kind animal) (ind ?x ?kind))
```

The implementation of this new language can be based directly on the previous implementation of dtrees. Each assertion is stored as a fact in a dtree, except that the components of an and assertion are stored separately. The function add-fact does this:

```
(defun add-fact (fact)
  "Add the fact to the data base."
  (if (eq (predicate fact) 'and)
      (mapc #'add-fact (args fact))
      (index fact)))
```

Querying this new data base consists of querying the dtree just as before, but with a special case for conjunctive (and) queries. Conceptually, the function to do this, retrieve-fact, should be as simple as the following:

---

```
(defun retrieve-fact (query)
  "Find all facts that match query. Return a list of bindings.
  Warning!! this version is incomplete."
  (if (eq (predicate query) 'and)
      (retrieve-conjunction (args query))
      (retrieve query bindings)))
```

Unfortunately, there are some complications. Think about what must be done in `retrieve-conjunction`. It is passed a list of conjuncts and must return a list of binding lists, where each binding list satisfies the query. For example, to find out what people were born on July 1st, we could use the query:

```
(and (val birthday ?p july-1) (ind ?p person))
```

`retrieve-conjunction` could solve this problem by first calling `retrieve-fact` on `(val birthday ?p july-1)`. Once that is done, there is only one conjunct remaining, but in general there could be several, so we need to call `retrieve-conjunction` recursively with two arguments: the remaining conjuncts, and the result that `retrieve-fact` gave for the first solution. Since `retrieve-fact` returns a list of binding lists, it will be easiest if `retrieve-conjunction` accepts such a list as its second argument. Furthermore, when it comes time to call `retrieve-fact` on the second conjunct, we will want to respect the bindings set up by the first conjunct. So `retrieve-fact` must accept a binding list as its second argument. Thus we have:

```
(defun retrieve-fact (query &optional (bindings no-bindings))
  "Find all facts that match query. Return a list of bindings."
  (if (eq (predicate query) 'and)
      (retrieve-conjunction (args query) (list bindings))
      (retrieve query bindings)))

(defun retrieve-conjunction (conjuncts bindings-lists)
  "Return a list of binding lists satisfying the conjuncts."
  (mapcan
   #'(lambda (bindings)
       (cond ((eq bindings fail) nil)
             ((null conjuncts) (list bindings))
             (t (retrieve-conjunction
                  (rest conjuncts)
                  (retrieve-fact
                   (subst-bindings bindings (first conjuncts))
                   bindings)))))))
  bindings-lists))
```

Notice that `retrieve` and therefore `mapc-retrieve` now also must accept a binding list. The changes to them are shown in the following. In each case the extra argument

is made optional so that previously written functions that call these functions without passing in the extra argument will still work.

```
(defun mapc-retrieve (fn query &optional (bindings no-bindings))
  "For every fact that matches the query,
  apply the function to the binding list."
  (dolist (bucket (fetch query))
    (dolist (answer bucket)
      (let ((new-bindings (unify query answer bindings)))
        (unless (eq new-bindings fail)
          (funcall fn new-bindings)))))

(defun retrieve (query &optional (bindings no-bindings))
  "Find all facts that match query.  Return a list of bindings."
  (let ((answers nil))
    (mapc-retrieve #'(lambda (bindings) (push bindings answers))
                  query bindings)
    answers))
```

Now add-fact and retrieve-fact comprise all we need to implement the language. Here is a short example where add-fact is used to add facts about bears and dogs, both as individuals and as species:

```
> (add-fact '(sub dog animal)) => T
> (add-fact '(sub bear animal)) => T
> (add-fact '(ind Fido dog)) => T
> (add-fact '(ind Yogi bear)) => T
> (add-fact '(val color Yogi brown)) => T
> (add-fact '(val color Fido golden)) => T
> (add-fact '(val latin-name bear ursidae)) => T
> (add-fact '(val latin-name dog canis-familiaris)) => T
```

Now retrieve-fact is used to answer three questions: What kinds of animals are there? What are the Latin names of each kind of animal? and What are the colors of each individual bear?

```
> (retrieve-fact '(sub ?kind animal))
(((?KIND . DOG))
 ((?KIND . BEAR)))

> (retrieve-fact '(and (sub ?kind animal)
                        (val latin-name ?kind ?latin)))
(((?LATIN . CANIS-FAMILIARIS) (?KIND . DOG))
 ((?LATIN . URSIDAE) (?KIND . BEAR)))
```

---

```
> (retrieve-fact '(and (ind ?x bear) (val color ?x ?c)))
((?C . BROWN) (?X . YOGI)))
```

### *Improvements*

There are quite a few improvements that can be made to this system. One direction is to provide different kinds of answers to queries. The following two functions are similar to `retrieve-matches` in that they return lists of solutions that match the query, rather than lists of possible bindings:

```
(defun retrieve-bagof (query)
  "Find all facts that match query.
  Return a list of queries with bindings filled in."
  (mapcar #'(lambda (bindings) (subst-bindings bindings query))
          (retrieve-fact query)))

(defun retrieve-setof (query)
  "Find all facts that match query.
  Return a list of unique queries with bindings filled in."
  (remove-duplicates (retrieve-bagof query) :test #'equal))
```

Another direction to take is to provide better error checking. The current system does not complain if a fact or query is ill-formed. It also relies on the user to input all facts, even those that could be derived automatically from the semantics of existing facts. For example, the semantics of `sub` imply that if `(sub bear animal)` and `(sub polar-bear bear)` are true, then `(sub polar-bear animal)` must also be true. This kind of implication can be handled in two ways. The typical Prolog approach would be to write rules that derive the additional `sub` facts by backward-chaining. Then every query would have to check if there were rules to run. The alternative is to use a *forward-chaining* approach, which caches each new `sub` fact by adding it to the data base. This latter alternative takes more storage, but because it avoids rederiving the same facts over and over again, it tends to be faster.

The following version of `add-fact` does error checking, and it automatically caches facts that can be derived from existing facts. Both of these things are done by a set of functions that are attached to the primitive operators. It is done in a data-driven style to make it easier to add new primitives, should that become necessary.

The function `add-fact` checks that each argument to a primitive relation is a nonvariable atom, and it also calls `fact-present-p` to check if the fact is already present in the data base. If not, it indexes the fact and calls `run-attached-fn` to do additional checking and caching:

```
(defparameter *primitives* '(and sub ind rel val))
```

```
(defun add-fact (fact)
  "Add the fact to the data base."
  (cond ((eq (predicate fact) 'and)
         (mapc #'add-fact (args fact)))
        ((or (not (every #'atom (args fact)))
             (some #'variable-p (args fact)))
            (not (member (predicate fact) *primitives*)))
         (error "Ill-formed fact: ~a" fact))
        ((not (fact-present-p fact))
         (index fact)
         (run-attached-fn fact)))
        t)

(defun fact-present-p (fact)
  "Is this fact present in the data base?"
  (retrieve fact))
```

The attached functions are stored on the operator's property list under the indicator `attached-fn`:

```
(defun run-attached-fn (fact)
  "Run the function associated with the predicate of this fact."
  (apply (get (predicate fact) 'attached-fn) (args fact)))

(defmacro def-attached-fn (pred args &body body)
  "Define the attached function for a primitive."
  '(setf (get ',pred 'attached-fn)
        #'(lambda ,args .,body)))
```

The attached functions for `ind` and `val` are fairly simple. If we know (`sub bear animal`), then when (`ind Yogi bear`) is asserted, we have to also assert (`ind Yogi animal`). Similarly, the values in a `val` assertion must be individuals of the categories in the relation's `rel` assertion. That is, if (`rel birthday animal date`) is a fact and (`val birthday Lee july-1`) is added, then we can conclude (`ind Lee animal`) and (`ind july-1 date`). The following functions add the appropriate facts:

```
(def-attached-fn ind (individual category)
  ;;= Cache facts about inherited categories
  (query-bind (?super) '(sub ,category ?super)
    (add-fact '(ind ,individual ,?super))))
```

```
(def-attached-fn val (relation ind1 ind2)
  ;;= Make sure the individuals are the right kinds
  (query-bind (?cat1 ?cat2) '(rel ,relation ?cat1 ?cat2)
    (add-fact '(ind ,ind1 ,?cat1))
    (add-fact '(ind ,ind2 ,?cat2))))
```

The attached function for `rel` simply runs the attached function for any individual of the given relation. Normally one would make all `rel` assertions before `ind` assertions, so this will have no effect at all. But we want to be sure the data base stays consistent even if facts are asserted in an unusual order.

```
(def-attached-fn rel (relation cat1 cat2)
  ;;= Run attached function for any IND's of this relation
  (query-bind (?a ?b) '(ind ,relation ?a ?b)
    (run-attached-fn '(ind ,relation ,?a ,?b))))
```

The most complicated attached function is for `sub`. Adding a fact such as (`sub bear animal`) causes the following to happen:

- All of `animal`'s supercategories (such as `living-thing`) become supercategories of all of `bear`'s subcategories (such as `polar-bear`).
- `animal` itself becomes a supercategory all of `bear`'s subcategories.
- `bear` itself becomes a subcategory of all of `animal`'s supercategories.
- All of the individuals of `bear` become individuals of `animal` and its supercategories.

The following accomplishes these four tasks. It does it with four calls to `index-new-fact`, which is used instead of `add-fact` because we don't need to run the attached function on the new facts. We do, however, need to make sure that we aren't indexing the same fact twice.

```
(def-attached-fn sub (subcat supercat)
  ;;= Cache SUB facts
  (query-bind (?super-super) '(sub ,super-sub ?super-super)
    (index-new-fact '(sub ,subcat ,?super-super)))
    (query-bind (?sub-sub) '(sub ?sub-sub ,subcat)
      (index-new-fact '(sub ,?sub-sub ,?super-super))))
    (query-bind (?sub-sub) '(sub ?sub-sub ,subcat)
      (index-new-fact '(sub ,?sub-sub ,super-sub))))
    ;;= Cache IND facts
    (query-bind (?super-super) '(sub ,super-sub ?super-super)
      (query-bind (?sub-sub) '(sub ?sub-sub ,super-sub)
        (query-bind (?ind) '(ind ?ind ,?sub-sub)
          (index-new-fact '(ind ,?ind ,?super-super)))))))
```

```
(defun index-new-fact (fact)
  "Index the fact in the data base unless it is already there."
  (unless (fact-present-p fact)
    (index fact)))
```

The following function tests the attached functions. It shows that adding the single fact (sub bear animal) to the given data base causes 18 new facts to be added.

```
(defun test-bears ()
  (clear-dtrees)
  (mapc #'add-fact
    '((sub animal living-thing)
      (sub living-thing thing) (sub polar-bear bear)
      (sub grizzly bear) (ind Yogi bear) (ind Lars polar-bear)
      (ind Helga grizzly)))
  (trace index)
  (add-fact '(sub bear animal))
  (untrace index))

> (test-bears)
(1 ENTER INDEX: (SUB BEAR ANIMAL))
(1 EXIT INDEX: T)
(1 ENTER INDEX: (SUB BEAR THING))
(1 EXIT INDEX: T)
(1 ENTER INDEX: (SUB GRIZZLY THING))
(1 EXIT INDEX: T)
(1 ENTER INDEX: (SUB POLAR-BEAR THING))
(1 EXIT INDEX: T)
(1 ENTER INDEX: (SUB BEAR LIVING-THING))
(1 EXIT INDEX: T)
(1 ENTER INDEX: (SUB GRIZZLY LIVING-THING))
(1 EXIT INDEX: T)
(1 ENTER INDEX: (SUB POLAR-BEAR LIVING-THING))
(1 EXIT INDEX: T)
(1 ENTER INDEX: (SUB GRIZZLY ANIMAL))
(1 EXIT INDEX: T)
(1 ENTER INDEX: (SUB POLAR-BEAR ANIMAL))
(1 EXIT INDEX: T)
(1 ENTER INDEX: (IND LARS LIVING-THING))
(1 EXIT INDEX: T)
(1 ENTER INDEX: (IND HELGA LIVING-THING))
(1 EXIT INDEX: T)
(1 ENTER INDEX: (IND YOGI LIVING-THING))
(1 EXIT INDEX: T)
(1 ENTER INDEX: (IND LARS THING))
(1 EXIT INDEX: T)
(1 ENTER INDEX: (IND HELGA THING))
```

```
(1 EXIT INDEX: T)
(1 ENTER INDEX: (IND YOGI THING))
(1 EXIT INDEX: T)
(1 ENTER INDEX: (IND LARS ANIMAL))
(1 EXIT INDEX: T)
(1 ENTER INDEX: (IND HELGA ANIMAL))
(1 EXIT INDEX: T)
(1 ENTER INDEX: (IND YOGI ANIMAL))
(1 EXIT INDEX: T)
(INDEX)
```

### *A Frame Language*

Another direction we can take is to provide an alternative syntax that will be easier to read and write. Many representation languages are based on the idea of *frames*, and their syntax reflects this. A frame is an object with slots. We will continue to use the same data base in the same format, but we will provide an alternative syntax that considers the individuals and categories as frames, and the relations as slots.

Here is an example of the frame syntax for individuals, which uses the operator `a`. Note that it is more compact than the equivalent notation using the primitives.

```
(a person (name Joe) (age 27)) ≡
(and (ind person1 person)
      (val name person1 Joe)
      (val age person1 27))
```

The syntax also allows for nested expressions to appear as the values of slots. Notice that the Skolem constant `person1` was generated automatically; an alternative is to supply a constant for the individual after the category name. For example, the following says that Joe is a person of age 27 whose best friend is a person named Fran who is 28 and whose best friend is Joe:

```
(a person p1 (name Joe) (age 27)
    (best-friend (a person (name Fran) (age 28)
                           (best-friend p1)))) ≡
(and (ind p1 person) (val name p1 joe) (val age p1 27)
      (ind person2 person) (val name person2 fran)
      (val age person2 28) (val best-friend person2 p1)
      (val best-friend p1 person2))
```

The frame syntax for categories uses the operator each. For example:

```
(each person (isa animal) (name person-name) (age integer)) ≡
(and (sub person animal)
  (rel name person person-name)
  (rel age person integer))
```

The syntax for queries is the same as for assertions, except that variables are used instead of the Skolem constants. This is true even when the Skolem constants are automatically generated, as in the following query:

```
(a person (age 27)) ≡ (AND (IND ?3 PERSON) (VAL AGE ?3 27))
```

To support the frame notation, we define the macros a and each to make assertions and ?? to make queries.

```
(defmacro a (&rest args)
  "Define a new individual and assert facts about it in the data base."
  '(add-fact ',(translate-exp (cons 'a args))))"

(defmacro each (&rest args)
  "Define a new category and assert facts about it in the data base."
  '(add-fact ',(translate-exp (cons 'each args))))"

(defmacro ?? (&rest queries)
  "Return a list of answers satisfying the query or queries."
  '(retrieve-setof
    ',(translate-exp (maybe-add 'and (replace-?-vars queries))
      :query))))
```

All three of these macros call on translate-exp to translate from the frame syntax to the primitive syntax. Note that an a or each expression is computing a conjunction of primitive relations, but it is also computing a *term* when it is used as the nested value of a slot. It would be possible to do this by returning multiple values, but it is easier to build translate-exp as a set of local functions that construct facts and push them on the local variable conjuncts. At the end, the list of conjuncts is returned as the value of the translation. The local functions translate-a and translate-each return the atom that represents the term they are translating. The local function translate translates any kind of expression, translate-slot handles a slot, and collect-fact is responsible for pushing a fact onto the list of conjuncts. The optional argument query-mode-p tells what to do if the individual is not provided in an a expression. If query-mode-p is true, the individual will be represented by a variable; otherwise it will be a Skolem constant.

```
(defun translate-exp (exp &optional query-mode-p)
  "Translate exp into a conjunction of the four primitives."
  (let ((conjuncts nil))
    (labels
      ((collect-fact (&rest terms) (push terms conjuncts))

       (translate (exp)
          ;;= Figure out what kind of expression this is
          (cond
            ((atom exp) exp)
            ((eq (first exp) 'a) (translate-a (rest exp)))
            ((eq (first exp) 'each) (translate-each (rest exp)))
            (t (apply #'collect-fact exp) exp)))

       (translate-a (args)
          ;;= translate (A category [ind] (rel filler)*)
          (let* ((category (pop args))
                 (self (cond ((and args (atom (first args)))
                               (pop args))
                            (query-mode-p (gentemp "?"))
                            (t (gentemp (string category)))))))
            (collect-fact 'ind self category)
            (dolist (slot args)
              (translate-slot 'val self slot))
            self))

       (translate-each (args)
          ;;= translate (EACH category [(isa cat*)] (slot cat)*)
          (let* ((category (pop args)))
            (when (eq (predicate (first args)) 'isa)
              (dolist (super (rest (pop args)))
                (collect-fact 'sub category super)))
            (dolist (slot args)
              (translate-slot 'rel category slot))
            category))

       (translate-slot (primitive self slot)
          ;;= translate (relation value) into a REL or SUB
          (assert (= (length slot) 2))
          (collect-fact primitive (first slot) self
                        (translate (second slot)))))

       ;;= Body of translate-exp:
       (translate exp) ;;= Build up the list of conjuncts
       (maybe-add 'and (nreverse conjuncts))))))
```

The auxiliary functions `maybe-add` and `replace-?-vars` are shown in the following:

```
(defun maybe-add (op exps &optional if-nil)
  "For example, (maybe-add 'and exps t) returns
  t if exps is nil, (first exps) if there is only one,
  and (and exp1 exp2...) if there are several exps."
  (cond ((null exps) if-nil)
        ((length=1 exps) (first exps))
        (t (cons op exps)))))

(defun length=1 (x)
  "Is x a list of length 1?"
  (and (consp x) (null (cdr x)))))

(defun replace-?-vars (exp)
  "Replace each ? in exp with a temporary var: ?123"
  (cond ((eq exp '?) (gentemp "?"))
        ((atom exp) exp)
        (t (reuse-cons (replace-?-vars (first exp))
                      (replace-?-vars (rest exp))
                      exp))))
```

### *Possible Worlds: Truth, Negation, and Disjunction*

In this section we address four problems: distinguishing unknown from `false`, representing negations, representing disjunctions, and representing multiple possible states of affairs. It turns out that all four problems can be solved by introducing two new techniques: possible worlds and negated predicates. The solution is not completely general, but it is practical in a wide variety of applications.

There are two basic ways to distinguish unknown from `false`. The first possibility is to store a truth value—`true` or `false`—along with each proposition. The second possibility is to include the truth value as part of the proposition. There are several syntactic variations on this theme. The following table shows the possibilities for the propositions “Jan likes Dean is true” and “Jan likes Ian is false.”

Approach	True Prop.	False Prop.
(1)	(likes Jan Dean) -- true	(likes Jan Ian) -- false
(2a)	(likes true Jan Dean)	(likes false Jan Ian)
(2b)	(likes Jan Dean)	(not (likes Jan Dean))
(2c)	(likes Jan Dean)	(~likes Jan Dean)

The difference between (1) and (2) shows up when we want to make a query. With (1), we make the single query `(likes Jan Dean)` (or perhaps `(likes Jan ?x)`), and the answers will tell us who Jan does and does not like. With (2), we make one

query to find out what liking relationships are true, and another to find out which ones are false. In either approach, if there are no responses then the answer is truly unknown.

Approach (1) is better for applications where most queries are of the form “Is this sentence true or false?” But applications that include backward-chaining rules are not like this. The typical backward-chaining rule says “Conclude X is true if Y is true.” Thus, most queries will be of the type “Is Y true?” Therefore, some version of approach (2) is preferred.

Representing true and false opens the door to a host of possible extensions. First, we could add multiple truth values beyond the simple “true” and “false.” These could be symbolic values like “probably-true” or “false-by-default” or they could be numeric values representing probabilities or certainty factors.

Second, we could introduce the idea of *possible worlds*. That is, the truth of a proposition could be unknown in the current world, but true if we assume  $p$ , and false if we assume  $q$ . In the possible world approach, this is handled by calling the current world  $W$ , and then creating a new world  $W_1$ , which is just like  $W$  except that  $p$  is true, and  $W_2$ , which is just like  $W$  except that  $q$  is true. By doing reasoning in different worlds we can make predictions about the future, resolve ambiguities about the current state, and do reasoning by cases.

For example, possible worlds allow us to solve Moore’s communism/democracy problem (page 466). We create two new possible worlds, one where  $E$  is a democracy and one where it is communist. In each world it is easy to derive that there is a democracy next to a communist country. The trick is to realize then that the two worlds form a partition, and that therefore the assertion holds in the original “real” world as well. This requires an interaction between the Prolog-based tactical reasoning going on within a world and the planning-based strategic reasoning that decides which worlds to consider.

We could also add a *truth maintenance system* (or TMS) to keep track of the assumptions or justifications that lead to each fact being considered true. A truth maintenance system can lessen the need to backtrack in a search for a global solution. Although truth maintenance systems are an important part of AI programming, they will not be covered in this book.

In this section we extend the dtree facility (section 14.8) to handle truth values and possible worlds. With so many options, it is difficult to make design choices. We will choose a fairly simple system, one that remains close to the simplicity and speed of Prolog but offers additional functionality when needed. We will adopt approach (2c) to truth values, using negated predicates. For example, the negated predicate of `likes` is `~likes`, which is pronounced “not likes.”

We will also provide minimal support for possible worlds. Assume that there is always a current world,  $W$ , and that there is a way to create alternative worlds and change the current world to an alternative one. Assertions and queries will always be made with respect to the current world. Each fact is indexed by the atoms it contains,

just as before. The difference is that the facts are also indexed by the current world. To support this, we need to modify the notion of the numbered list, or `nlist`, to include a numbered association list, or `nalist`. The following is an `nalist` showing six facts indexed under three different worlds: `W0`, `W1`, and `W2`:

```
(6 (W0 #1# #2# #3#) (W1 #4#) (W2 #5# #6#))
```

The fetching routine will remain unchanged, but the postfetch processing will have to sort through the `nalists` to find only the facts in the current world. It would also be possible for `fetch` to do this work, but the reasoning is that most facts will be indexed under the “real world,” and only a few facts will exist in alternative, hypothetical worlds. Therefore, we should delay the effort of sorting through the answers to eliminate those answers in the wrong world—it may be that the first answer fetched will suffice, and then it would have been a waste to go through and eliminate other answers. The following changes to `index` and `dtree-index` add support for worlds:

```
(defvar *world* 'W0 "The current world used by index and fetch.")

(defun index (key &optional (world *world*))
  "Store key in a dtree node. Key must be (predicate . args);
  it is stored in the dtree, indexed by the world."
  (dtree-index key key world (get-dtree (predicate key)))))

(defun dtree-index (key value world dtree)
  "Index value under all atoms of key in dtree."
  (cond
    ((consp key) ; index on both first and rest
     (dtree-index (first key) value world
                  (or (dtree-first dtree)
                      (setf (dtree-first dtree) (make-dtree))))
     (dtree-index (rest key) value world
                  (or (dtree-rest dtree)
                      (setf (dtree-rest dtree) (make-dtree)))))
    ((null key)) ; don't index on nil

    ((variable-p key) ; index a variable
     (nalist-push world value (dtree-var dtree)))
    (t ; Make sure there is an nlist for this atom, and add to it
     (nalist-push world value (lookup-atom key dtree)))))
```

The new function `nalist-push` adds a value to an `nalist`, either by inserting the value in an existing key’s list or by adding a new key/value list:

---

```
(defun nalist-push (key val nalist)
  "Index val under key in a numbered alist."
  ;; An nalist is of the form (count (key val*)*)
  ;; Ex: (6 (nums 1 2 3) (letters a b c))
  (incf (car nalist))
  (let ((pair (assoc key (cdr nalist))))
    (if pair
        (push val (cdr pair))
        (push (list key val) (cdr nalist)))))
```

In the following, `fetch` is used on the same data base created by `test-index`, indexed under the world `W0`. This time the result is a list-of-lists of world/values a-lists. The count, 3, is the same as before.

```
> (fetch '(p ?x c))
(((W0 (P B C) (P A C)))
 ((W0 (P A ?X))))
 3
```

So far, worlds have been represented as symbols, with the implication that different symbols represent completely distinct worlds. That doesn't make worlds very easy to use. We would like to be able to use worlds to explore alternatives—create a new hypothetical world, make some assumptions (by asserting them as facts in the hypothetical world), and see what can be derived in that world. It would be tedious to have to copy all the facts from the real world into each hypothetical world.

An alternative is to establish an inheritance hierarchy among worlds. Then a fact is considered true if it is indexed in the current world or in any world that the current world inherits from.

To support inheritance, we will implement worlds as structures with a `name` field and a field for the list of parents the world inherits from. Searching through the inheritance lattice could become costly, so we will do it only once each time the user changes worlds, and mark all the current worlds by setting the `current` field on or off. Here is the definition for the `world` structure:

```
(defstruct (world (:print-function print-world))
  name parents current)
```

We will need a way to get from the name of a world to the `world` structure. Assuming names are symbols, we can store the structure on the name's property list. The function `get-world` gets the structure for a name, or builds a new one and stores it. `get-world` can also be passed a world instead of a name, in which case it just returns the world. We also include a definition of the default initial world.

```
(defun get-world (name &optional current (parents (list *world*)))
  "Look up or create the world with this name.
  If the world is new, give it the list of parents."
  (cond ((world-p name) name) ; ok if it already is a world
        ((get name 'world))
        (t (setf (get name 'world)
                  (make-world :name name :parents parents
                               :current current)))))

(defvar *world* (get-world 'W0 nil nil)
  "The current world used by index and fetch.")
```

The function `use-world` is used to switch to a new world. It first makes the current world and all its parents no longer current, and then makes the new chosen world and all its parents current. The function `use-new-world` is more efficient in the common case where you want to create a new world that inherits from the current world. It doesn't have to turn any worlds off; it just creates the new world and makes it current.

```
(defun use-world (world)
  "Make this world current."
  ;;= If passed a name, look up the world it names
  (setf world (get-world world))
  (unless (eq world *world*)
    ;;= Turn the old world(s) off and the new one(s) on,
    ;;= unless we are already using the new world
    (set-world-current *world* nil)
    (set-world-current world t)
    (setf *world* world)))

(defun use-new-world ()
  "Make up a new world and use it.
  The world inherits from the current world."
  (setf *world* (get-world (gensym "W")))
  (setf (world-current *world*) t)
  *world*)

(defun set-world-current (world on/off)
  "Set the current field of world and its parents on or off."
  ;;= nil is off, anything else is on.
  (setf (world-current world) on/off)
  (dolist (parent (world-parents world))
    (set-world-current parent on/off)))
```

We also add a print function for worlds, which just prints the world's name.

---

```
(defun print-world (world &optional (stream t) depth)
  (declare (ignore depth))
  (prin1 (world-name world) stream))
```

The format of the dtree data base has changed to include worlds, so we need new retrieval functions to search through this new format. Here the functions mapc-retrieve, retrieve, and retrieve-bagof are modified to give new versions that treat worlds. To reflect this change, the new functions all have names ending in -in-world:

```
(defun mapc-retrieve-in-world (fn query)
  "For every fact in the current world that matches the query,
  apply the function to the binding list."
  (dolist (bucket (fetch query))
    (dolist (world/entries bucket)
      (when (world-current (first world/entries))
        (dolist (answer (rest world/entries))
          (let ((bindings (unify query answer)))
            (unless (eq bindings fail)
              (funcall fn bindings)))))))

(defun retrieve-in-world (query)
  "Find all facts that match query.  Return a list of bindings."
  (let ((answers nil))
    (mapc-retrieve-in-world
      #'(lambda (bindings) (push bindings answers))
      query)
    answers))

(defun retrieve-bagof-in-world (query)
  "Find all facts in the current world that match query.
  Return a list of queries with bindings filled in."
  (mapcar #'(lambda (bindings) (subst-bindings bindings query))
    (retrieve-in-world query)))
```

Now let's see how these worlds work. First, in W0 we see that the facts from test-index are still in the data base:

```
> *world* => W0
> (retrieve-bagof-in-world '(p ?z c)) =>
((P A C) (P A C) (P B C))
```

Now we create and use a new world that inherits from W0. Two new facts are added to this new world:

```
> (use-new-world) => W7031
> (index '(p new c)) => T
> (index '(~p b b)) => T
```

We see that the two new facts are accessible in this world:

```
> (retrieve-bagof-in-world '(p ?z c)) =>
((P A C) (P A C) (P B C) (P NEW C))
> (retrieve-bagof-in-world '(~p ?x ?y)) =>
((~P B B))
```

Now we create another world as an alternative to the current one by first switching back to the original W0, then creating the new world, and then adding some facts:

```
> (use-world 'W0) => W0
> (use-new-world) => W7173
> (index '(p newest c)) => T
> (index '(~p c newest)) => T
```

Here we see that the facts entered in W7031 are not accessible, but the facts in the new world and in W0 are:

```
> (retrieve-bagof-in-world '(p ?z c)) =>
((P A C) (P A C) (P B C) (P NEWEST C))
> (retrieve-bagof-in-world '(~p ?x ?y)) =>
((~P C NEWEST))
```

### *Unification, Equality, Types, and Skolem Constants*

The lesson of the zebra puzzle in section 11.4 was that unification can be used to lessen the need for backtracking, because an uninstantiated logic variable or partially instantiated term can stand for a whole range of possible solutions. However, this advantage can quickly disappear when the representation forces the problem solver to enumerate possible solutions rather than treating a whole range of solutions as one. For example, consider the following query in the frame language and its expansion into primitives:

---


$$\begin{aligned}
 & (\text{a person} (\text{name Fran})) \\
 & \equiv (\text{and} (\text{ind ?p person}) (\text{val name ?p fran}))
 \end{aligned}$$

The way to answer this query is to enumerate all individuals  $?p$  of type `person` and then check the `name` slot of each such person. It would be more efficient if `(ind ?p person)` did not act as an enumeration, but rather as a constraint on the possible values of  $?p$ . This would be possible if we changed the definition of variables (and of the unification function) so that each variable had a type associated with it. In fact, there are at least three sources of information that have been implemented as constraints on variables terms:

- The type or category of the term.
- The members or size of a term considered as a set or list.
- Other terms this term is equal or not equal to.

Note that with a good solution to the problem of equality, we can solve the problem of Skolem constants. The idea is that a regular constant unifies with itself but no other regular constant. On the other hand, a Skolem constant can potentially unify with any other constant (regular or Skolem). The equality mechanism is used to keep track of each Skolem variable's possible bindings.

## 14.11 History and References

Brachman and Levesque (1985) collect thirty of the key papers in knowledge representation. Included are some early approaches to semantic network based (Quillian 1967) and logic-based (McCarthy 1968) representation. Two thoughtful critiques of the ad hoc use of representations without defining their meaning are by Woods (1975) and McDermott (1978). It is interesting to contrast the latter with McDermott 1987, which argues that logic by itself is not sufficient to solve the problems of AI. This argument should not be surprising to those who remember the slogan *logic = algorithm – control*.

Genesereth and Nilsson's textbook (1987) cover the predicate-calculus-based approach to knowledge representation and AI in general. Ernest Davis (1990) presents a good overview of the field that includes specialized representations for time, space, qualitative physics, propositional attitudes, and the interaction between agents.

Many representation languages focus on the problem of defining descriptions for categories of objects. These have come to be known as *term-subsumption languages*. Examples include KL-ONE (Schmolze and Lipkis 1983) and KRYPTON (Brachman, Fikes, and Levesque 1983). See Lakoff 1987 for much more on the problem of categories and prototypes.

Hector Levesque (1986) points out that the areas Prolog has difficulty with—disjunction, negation, and existentials—all involve a degree of vagueness. In his term, they lack *vividness*. A vivid proposition is one that could be represented directly in a picture: the car is blue; she has a martini in her left hand; Albany is the capital of New York. Nonvivid propositions cannot be so represented: the car is not blue; she has a martini in one hand; either Albany or New York City is the capital of New York. There is interest in separating vivid from nonvivid reasoning, but no current systems are actually built this way.

The possible world approach of section 14.10 was used in the MRS system (Russell 1985). More recent knowledge representation systems tend to use truth maintenance systems instead of possible worlds. This approach was pioneered by Doyle (1979) and McAllester (1982). Doyle tried to change the name to “reason maintenance,” in (1983), but it was too late. The version in widest use today is the assumption-based truth maintenance system, or ATMS, developed by de Kleer (1986a,b,c). Charniak et al. (1987) present a complete Common Lisp implementation of a McAllester-style TMS.

There is little communication between the logic programming and knowledge representation communities, even though they cover overlapping territory. Colmerauer (1990) and Cohen (1990) describe Logic Programming languages that address some of the issues covered in this chapter. Key papers in equality reasoning include Galler and Fisher 1974, Kornfeld 1983,<sup>1</sup> Jaffar, Lassez, and Maher 1984, and van Emden and Yukawa 1987. Hölldobler’s book (1987) includes an overview of the area. Papers on extending unification in ways other than equality include Aït-Kaci et al. 1987 and Staples and Robinson 1988. Finally, papers on extending Prolog to cover disjunction and negation (i.e., non-Horn clauses) include Loveland 1987, Plaisted 1988, and Stickel 1988.

## 14.12 Exercises

-  **Exercise 14.1 [m]** Arrange to store dtrees in a hash table rather than on the property list of predicates.
-  **Exercise 14.2 [m]** Arrange to store the dtree-atoms in a hash table rather than in an association list.
-  **Exercise 14.3 [m]** Change the dtree code so that `nil` is used as an atom index. Time the performance on an application and see if the change helps or hurts.

---

<sup>1</sup>A commentary on this paper appears in Elcock and Hoddinott 1986.

- ❑ **Exercise 14.4 [m]** Consider the query `(p a b c d e f g)`. If the index under `a` returns only one or two keys, then it is probably a waste of time for `dtree-fetch` to consider the other keys in the hope of finding a smaller bucket. It is certainly a waste if there are no keys at all indexed under `a`. Make appropriate changes to `dtree-fetch`.
- ❑ **Exercise 14.5 [h]** Arrange to delete elements from a `dtree`.
- ❑ **Exercise 14.6 [h]** Implement iterative-deepening search in the Prolog compiler. You will have to change each function to accept the depth as an extra argument, and compile in checks for reaching the maximum depth.
- ❑ **Exercise 14.7 [d]** Integrate the Prolog compiler with the `dtree` data base. Use the `dtrees` for predicates with a large number of clauses, and make sure that each predicate that is implemented as a `dtree` has a Prolog primitive accessing the `dtree`.
- ❑ **Exercise 14.8 [d]** Add support for possible worlds to the Prolog compiler with `dtrees`. This support has already been provided for `dtrees`, but you will have to provide it for ordinary Prolog rules.
- ❑ **Exercise 14.9 [h]** Integrate the language described in section 14.10 and the frame syntax from section 14.10 with the extended Prolog compiler from the previous exercise.
- ❑ **Exercise 14.10 [d]** Build a strategic reasoner that decides when to create a possible world and does reasoning by cases over these worlds. Use it to solve Moore's problem (page 466).

## 14.13 Answers

### Answer 14.1

```
(let ((dtrees (make-hash-table :test #'eq)))

  (defun get-dtree (predicate)
    "Fetch (or make) the dtree for this predicate."
    (setf (gethash predicate dtrees)
          (or (gethash predicate dtrees)
              (make-dtree)))

  (defun clear-dtrees ()
    "Remove all the dtrees for all the predicates."
    (clrhash dtrees)))
```

**Answer 14.5** Hint: here is the code for `nlist-delete`. Now figure out how to find all the nlists that an item is indexed under.

```
(defun nlist-delete (item nlist)
  "Remove an element from an nlist.
Assumes that item is present exactly once."
  (decf (car nlist))
  (setf (cdr nlist) (delete item (cdr nlist) :count 1))
  nlist)
```