SYMBOLIC COMPUTING WITH LISP AND PROLOG

ROBERT A. MUELLER REX L. PAGE

SYMBOLIC COMPUTING WITH LISP AND PROLOG

Robert A. Mueller
Colorado State University, Quantitative Technology Corporation

Rex L. Page
Colorado State University, Amoco Research

WILEY
JOHN WILEY & SONS
New York • Chichester • Brisbane • Toronto • Singapore

To the memories of Walter Orvedahl and Barney Marschner, mentors extraordinaire. Walter, you always knew the right place to go, and Barney, you always knew how to get there.

Cover photograph by Geoffry Gove

Unix is a trademark of AT&T Bell Laboratories

Library of Congress Cataloging in Publication Data:

Mueller, Robert A.
Symbolic Computing with Lisp and Prolog / Robert A. Mueller, Rex L. Page
p. cm.
ISBN 0-471-60771-1
1. LISP (Computer program language) 2. Prolog (Computer program language) I. Page, Rex L. II. Title.
QA76.73.L23M84 1988
005.13'3--dc19 88-23414
CIP

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

SYMBOLIC COMPUTING WITH LISP AND PROLOG

PREFACE

Two main themes emerge in this book: symbolic computing and denotational programming. The first portions of the text covers programming, and the later portion discusses symbolic computing in such areas as game playing, language translation, and theorem proving. We present example problems in each of these areas, propose solutions, and then specify working programs in terms of the programming techniques and languages discussed in the first portion of the text.

Two programs accompany each of the applications in symbolic computing—one written in Lisp, the other in Prolog. These stylized programs conform to a collection of programming techniques that we classify as *denotational* in nature. By this we mean that they specify their results directly in terms of their input data. At each level in the specification, results are described entirely in terms of the constituents of the input data at that level. This contrasts with the *operational* approach, in which a sequence of operations, individually operating on their own small parts of the data in a step-by-step fashion, generates the desired output. To use one of the most overworked phrases of our time, denotational programs concentrate on *what* the result is, whereas operational programs emphasize *how* the result is computed.

To illustrate the difference between the denotational and operational forms, consider the following definition of *bread* (slightly paraphrased from *Webster's New Collegiate Dictionary*[1]):

> A leavened and baked food made of a mixture whose basic constituent is flour or meal.

[1] *Webster's New Collegiate Dictionary*, G. & C. Merriam, Springfield, Mass., 1974.

We would consider this definition to be denotational; the characteristic properties of bread are explicit, and the method of making it is implicit. Contrast this with an operational definition, which explicitly delineates the method of construction but leaves the resulting properties implicit:

1. Prepare the dough by mixing the flour, yeast in warm water, etc.
2. Knead the dough with a folding and tearing motion.
3. Allow the dough to recover by placing it in a warm, draft-free place for the required time.
4. Reknead the dough, shape accordingly, and allow it to rise again.
5. Bake in the oven for the required time at a suitable temperature.
6. Remove from the oven and allow to cool on a wire rack.

Denotational programs written in Lisp are *functional programs*: They describe the result as a function of the input; given a particular input, the Lisp system computes the result satisfying the functional specification. In Prolog they are *relational programs*: They describe constraints among the constituents of the input data and the results; combinations of input data and results satisfying these constraints are uncovered by the Prolog system and delivered as output. (There is a close correspondence between functional and relational programs. A function can be thought of as a constraint between potential input data and potential results that associates valid results with appropriate input. A relation does the same thing, although it has a bit more freedom in this regard than does a function, according to the technical definitions of the terms.) In either case, a computer system can derive computations from these specifications. We do not dwell on how such a system is able to do this, but we explain enough about these mechanisms to estimate the computational resources required by programs and to facilitate coping with bugs when they creep in.

Our approach is practical rather than theoretical. We emphasize useful, common programming techniques and show how they can affect the performance of a program. We value algorithms that avoid excessive computation. (We do not present $O(n^2)$ algorithms when linear or $O(n \log n)$ algorithms are available.) However, we do not attempt to pin down details of performance relative to specific configurations of computing hardware. We value programs that elucidate as well as compute, as this, we suspect, is a way out of the software quagmire, if there is one. (Paradoxically, it may also foster the reduction of hardware deficiencies, via massive parallelism, but that is not a primary motif in this presentation.)

This text can be used in several ways: to study some important applications in the area of symbolic computing, to practice techniques of

denotational programming in Lisp or in Prolog, or to learn about any combination of these applications and programming techniques. The Lisp portion and the Prolog portion are independent; neither part assumes a knowledge of the other. We present symbolic-computing algorithms in both notations, so that a familiarity with either Lisp or Prolog will be sufficient background for a study of that material. Readers who are already familiar with one of the languages may wish to skip both the programming sections and read only the symbolic computing portion of the text.

Whatever your route, we hope you will enjoy it and find the information useful.

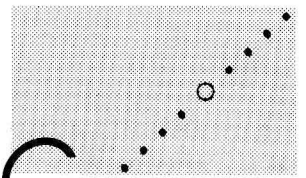
ROBERT A. MUELLER
REX L. PAGE

Acknowledgments

Many students persevered through the errors and organizational mistakes in earlier versions of this text and managed, in spite of these, to respond enthusiastically to the subject. We thank them for that, and for the guidance their reactions provided in revisions. Marian Sexton and Charles Sharpe provided an extraordinary number of detailed tips, for which we owe them a great debt. Margaret Sweeney and Joseph Varghese read the original manuscript with a critical eye when few others had seen it. We have some inkling of how hard that job was, and we very much appreciate their efforts.

Several reviewers saw value in the text and offered suggestions that led to important improvements in both context and presentation. We offer them our thanks for their distinctive contributions: Alan Perlis of Yale University for his inspiring endorsement of the concept of the book; David Touretsky of Carnegie-Mellon University for suggesting ways to connect the material to traditional presentations; Frederick Blackwell of California State University, Sacramento; Charles Dyer of the University of Wisconsin; Dennis Kibler of the University of California, Irvine; and Jordan Pollack of New Mexico State University for improving the organization of the text; and Richard Gabriel of Lucid Corporation and Stanford University for helping us reduce the similarity between our prose and pompous imitations of nineteenth-century novelists. Thank you every one.

CONTENTS



CHAPTER 1

LISP, PROLOG, AND DENOTATIONAL PROGRAMMING

Lisp and Prolog are notations for describing relationships between input data and results. They are general-purpose notations in the sense that they can be used to describe any such relationship that is "computable."

What is computable and what is not computable is a philosophical issue that has been debated for about half a century. Although there is no firm answer to the question, most of the discussion falls along these lines: A relationship between input and results is computable if it has a finite description in terms of a finite set of fundamental relationships, each of which can be worked out in a finite amount of time. The set of "fundamental relationships" is a key issue. Philosophers have considered several different fundamental sets (also known as "models of computation") and have found, through careful mathematical analysis, that they all are equivalent. That is, all the proposed models of computation lead to the same set of computable relationships. Lisp and Prolog are complete models in the sense that they are sufficient to describe any computable relationship that can be described via any of the other proposed methods of computation.

There are many other notations for describing computable relationships. You are probably familiar with one or more of them, such as Pascal, Fortran, C, PL/I, Cobol, or Basic. All these notations, or "programming languages" as they are usually called, start from slightly different sets of fundamental operations and from widely different points of view about how these operations should be denoted. They are all highly redundant in the sense that many of their fundamental operations could be eliminated without affecting the set of describable relationships.

Some of the redundancy leads to more concise programs. (A "program" for our purposes is a description in a programming language of a computable relationship.) Other portions of the redundancy are included to make it possible to address certain types of computing hardware in a particularly efficient manner. We will be concerned with both conciseness and efficiency, but we will not be overly concerned with questions of efficient computation that are closely related to underlying hardware.

For this reason, we will not discuss every aspect of Lisp and Prolog. We will cover subsets that are sufficient to describe any computable relationship in such a way that a computer can carry out the computation in a reasonably efficient manner, but not necessarily in the manner that is optimal with respect to a given computer's unique capabilities. The advantage of this approach is that you can spend more time learning general programming techniques and solutions for many computing applications. The disadvantage is that you will not learn all the bells and whistles of either Lisp or Prolog. If that is your goal, you should choose a different book; there are many suitable ones available.

In any programming language, even one that has no built-in redundancy, many different programs describe any given computable relationship. Although it makes us guilty of a vast oversimplification, we classify programs in two categories: *operational* and *denotational*. A program in the operational category describes a computable relationship in terms of a sequence of operations on the input data that eventually leads, in a step-by-step fashion, to the desired results. Thus, an operational program explains a procedure for computing the result. A denotational program, on the other hand, attempts to describe the form of the result in terms of its relationship to the input data. It specifies this relationship directly in terms of the constituents of the input data rather than concentrating on how the result can be derived from the input data in a step-by-step procedure.

To say the same thing one more way, in an operational program a procedure determines a relationship between input and output data; in a denotational program a relationship between input and output data determines a procedure. [1]

[1]The terms *iterative* and *procedural* often refer to the operational approach to programming. Names used for the denotational approach include *functional, applicative, declarative,* and *nonprocedural*. People often associate the terms functional and applicative with Lisp and declarative and nonprocedural with Prolog. We chose the term denotational because

For example, suppose we want to write a program that has as its result the word **yes** if its input data is a palindrome, and **no** if it is not. A denotational description might say that the result is **yes** whenever the letters in the input match, exactly, those same letters in reverse order; in all other situations, the result would be **no**. An operational description might go as follows: (1) Compare the first letter in the input data to the last letter; (2) if they are different, then the result is **no**; (3) if they are the same, then compare the second letter to the next-to-last; (4) if they are different, then the result is **no**; (5) continue in this manner until you have examined all the letters and found no differences, in which case the result is **yes**, or until you have determined that the result is **no**. The subsets of Lisp and Prolog that we will discuss favor denotational programs over operational programs.

Lisp was inspired by one of the models of computation that is denotational in nature, specifically, the lambda-calculus model proposed by Alonzo Church in the 1930s and further developed, in various forms, by Schoenfinkel, Curry, and others. In the 1950s, John McCarthy took the lead in developing the original Lisp notation and its supporting computing system, which was a practical implementation of many of the theoretical concepts developed by Church. (We write our programs in Common Lisp, a modern Lisp dialect. If you use a different dialect, you may have to adjust our notation slightly to get your programs to work. We stay within a very small subset of the language; few of the features we discuss will differ in any Lisp dialect.)

Prolog derives from the work of Frege on predicate calculus around the turn of the century, with subsequent refinements by Skolem, Herbrand, Horn, Davis and Putnam, Gilmore, and Robinson (spanning from the 1920s into the 1960s). The computational ramifications of this model were developed by Kowalski, Colmerauer, and others in the 1970s. (We write our Prolog programs in C-Prolog. If your Prolog system plays in a different key, you will have to transpose; we do not think you will find it difficult.)

Because Lisp and Prolog arose from descriptive models of computable relationships, they support the design of denotational programs more directly than do conventional programming languages such as Pascal or C, which are patterned after an operational model of computation developed by Alan Turing, Emil Post, and others. (The Turing model arose at about the same time as the Church model.)

Lisp and Prolog are not the only choices. There are many existing notations that would serve as a basis for illustrating the programming tech-

its dictionary meaning matches the idea of programs denoting results rather than operation sequences better than functional (connotes something that is not broken), applicative (useful for a given purpose), or declarative ("well, I declare"). Nonprocedural works, but seems too negative. Semantic specification techniques split naturally into denotational and operational categories, for many of the same reasons as programming, and both terms have gained some popularity in that field.

niques and applications covered in this text. For example, we could have chosen to use the mathematical notation of recursive function theory, or our own variant thereof, or we could have chosen Church's original notation, or Curry's, or Schoenfinkel's, or Skolem's, or Horn's.

The primary disadvantage of choosing one of these notations would be that our programs would stand only as descriptions of computations. They would not support, in a realistic sense, actual computations because we would not have access to a computer system that would automatically carry out the computations described by our programs.

That eliminates notations with no supporting computing system, but there are also many programming languages with excellent support for our chosen computational model that do have computer systems to support them. FP, the variableless programming language introduced by John Backus in the late 1970s would have been a good choice, as would SASL, KRC, or Miranda, the elegant notations developed by David Turner. ML, developed by Robin Milner's group, and Hope, developed by Burstall and associates, ALFL (Paul Hudak), and a host of new arrivals provide still other pleasing alternatives. We did not choose any of these because they are not so widely available as Lisp and Prolog. If you write a program in Lisp or Prolog, there will be many more places that you can use it to perform computations than there will be if you write it in Miranda or Hope.

So the choice of Lisp and Prolog was a pragmatic one: they are adequate, and they are popular — and likely to remain so for a long time.

Lisp can serve as a durable, generic medium for expressing the ideas of modern functional and logic-based (relational) languages. Alan Perlis likes to think of Lisp as the machine language for the programming languages of the future and of Prolog, FP, Miranda, and such as the initial prototypes of these languages. We agree.

Lisp has facilities for dealing effectively with computer architectures as they exist today. In fact, most Lisp programs developed over the past two decades have an operational bent. (Analysis of significant artificial intelligence codes written in Lisp reveals that setq's, rplaca's, and the like account for about 90% of all the function references in such programs. These operations support procedural programming; we do not cover them in this text.) New architectures aimed at supporting the most important parts of Lisp directly have had a commercial presence for half a decade. Prolog machines, SASL machines, and the like build on this experience.

The newer programming languages will facilitate the discovery of the most appropriate language features for effective programming. Lisp may then absorb these features and continue its dominance. Or Prolog may impart such a strong influence on the ideas and products stimulated by Japan's fifth-generation computing effort that programmers will migrate in the Prolog direction. Or later arrivals such as FP or Miranda or Hope may somehow gain a foothold. (Probably none of the above, predictions being what they are.) Regardless of the chosen notation, we believe that

the denotational approach will play an increasingly significant role in program design and development.

We cover only a very small part of Lisp in this text. We cover a larger percentage of Prolog (it is smaller than Lisp). The portions of the two languages that we present, and the ways in which we use them, may leave the impression that differences between them are primarily superficial. This is not the case. We simply choose to write all our programs in a denotational form (or nonprocedural, or declarative, or whichever term you prefer), and this minimizes the gulf between Lisp and Prolog.

To illustrate the similarity between Lisp and Prolog, as we use them, consider a pair of specifications of the palindrome relationship. The specification that follows on the left is in the manner of Lisp, and the one on the right is in the manner of Prolog (expressed in a form more akin to informal mathematical notation than to the formal syntax of either Lisp or Prolog).

Lisp-like	**Prolog-like**
p(x) is x=rev(x)	p(x) if rev(x,x)
rev([]) is []	rev([],[])
rev(w^x) is append(rev(x),[w])	rev(w^x,y) if rev(x,z) and append(z,[w],y)

In the Lisp-like program, p, rev, and append are functions (they deliver transformed versions of their input values). In the Prolog-like programs, p, rev, and append are relations (expressing true or false conditions, depending on their input values). In both programs x, y, and z are phrases, w is a letter from the alphabet, square brackets enclose sequences, and the circumflex represents a sequence formed from an initial component (on the left of the circumflex) and another sequence (on the right).

The Lisp-like program says that the palindrome function is true if its argument is the same as a reversed copy of its argument. It further specifies that the reverse function, when applied to the empty sequence, delivers the empty sequence; when applied to a sequence beginning with the component w and consisting of the components in x following w, the reverse function delivers a sequence constructed by concatenating a reversed copy of x and the sequence with w as its only component.

The Prolog-like program says that a sequence satisfies the palindrome property if it is in a reversed relationship with itself. It further specifies that the empty sequence satisfies the reverse property with itself and that a sequence whose first component is w and whose following components are those of the sequence x satisfies the reverse property with a sequence y if there is a sequence z that satisfies the reverse property with x and if that sequence z together with the sequence with w as its only component satisfies the append property with y.

In case you are curious, the programs that follow describe these same computations in Lisp, Prolog, and C syntax.

Lisp

```
(defun p(x)
   (equal x (rev x)))
(defun rev(x)
   (if (null x)
       x
       (append (rev (cdr x))
               (list (car x)))))
```

Prolog

```
p(X) :- rev(X,X).
rev([],[]).
rev([W|X],Y) :- rev(X,Z),
                append(Z,[W],Y).
```

C

```
#define False  0
#define True   1
p(x,n)
char x[];
int n;
{int k;
 for (k=0; k<n; k++)
    if (x[k] <> x[n-k-1])
       return(False);
 return(True);
}
```

The C version employs the conventional procedural paradigm. It specifies a sequence of computational steps, leaving the resulting input/output relationship to be deduced from an understanding of the resulting process. The Lisp and Prolog versions, on the other hand, specify the desired input/output relationship, leaving the computational process to be deduced by the processor. For us, Lisp and Prolog serve the same purpose: They provide a practical means of expressing computations in enlightening ways.

■ Bibliography Notes

In the 1930s and 1940s, logicians explored many of the theoretical foundations of computing. A line of investigation based on lambda-calculus and the related combinatory calculus, pioneered by Church, Curry, Kleene, Rosser, Schoenfinkel, and others inspired practical computing systems for functional programming developed by McCarthy, Landin, and others

in the 1950s and 1960s. (Rosser prepared an enlightening history of this work in 1982.) Elegant extensions of these computing systems, have emerged from work by Backus, Burstall, Milner, Turner, and others in the 1970s and 1980s.

Another line of investigation with its roots in predicate calculus and with theoretical contributions from Herbrand, Horn, Skolem, and others in the 1930s, 1940s, and 1950s has led to computing systems that support relational programming grounded in the work of Kowalski, Colmerauer, Robinson, and others in the 1970s. In the 1980s, Clocksin and Mellish provided an implementation of this approach that is practical for computing purposes.

Conventional computing systems, both hardware and software, from the 1950s to the present, have followed a line of investigation pioneered by Turing, Post, and others whose theoretical formulations had more obvious representations in the form of constructable, physical machines than either lambda-calculus or predicate calculus.

John Backus (1978). Can programming be liberated from the von Neumann style? *Comm. ACM* **21**,(8) 613–641.

R. M. Burstall, D. B. MacQueen, and D. T. Sannella (1980). HOPE: An experimental applicative language. *ACM Lisp Conference* (August), 136–143.

Alonzo Church (1932). A set of postulates for the foundation of logic. *Annals Mathematics* **33** (2): 346–366.

Alonzo Church (1936). An unsolvable problem of elementary number theory. *American J. Mathematics* **58**, 345–363.

Alonzo Church and J. Barkley Rosser (1936). Some properties of conversion. *Trans. American Mathematical Society* **39**, 472–482.

Alonzo Church (1941). The calculi of lambda-conversion. *Annals of Mathematical Studies 6*, Princeton University Press, Princeton N.J.

W. F. Clocksin and C. S. Mellish (1981). *Programming in Prolog*, Springer-Verlag, New York.

A. Colmerauer (1973). Les systemes-Q our un formalisme pour analyser et synthetiser des phrases sur ordinateur. *Publication Interne No. 43*, Dept d'Informatique, University of Montreal, 1973.

A. Colmerauer (1978). Metamorphosis grammars. In L. Bolc (ed.), *Natural Language Communication with Computers*, Lecture Notes in Computer Science, Vol. 63. Springer-Verlag, New York.

Haskell B. Curry (1930). Grundlagen der Kombinatorischen Logik. *American J. of Mathematics* **52** 509–536, 789–834.

Haskell B. Curry (1963). *Foundations of Mathematical Logic*, McGraw-Hill, New York.

M. Gordon, R. Milner, and C. Wadsworth (1979). Edinburgh LCF – a mechanized logic of computation. *Lecture Notes in Computer Sciences*, Vol 78. Springer-Verlag, New York.