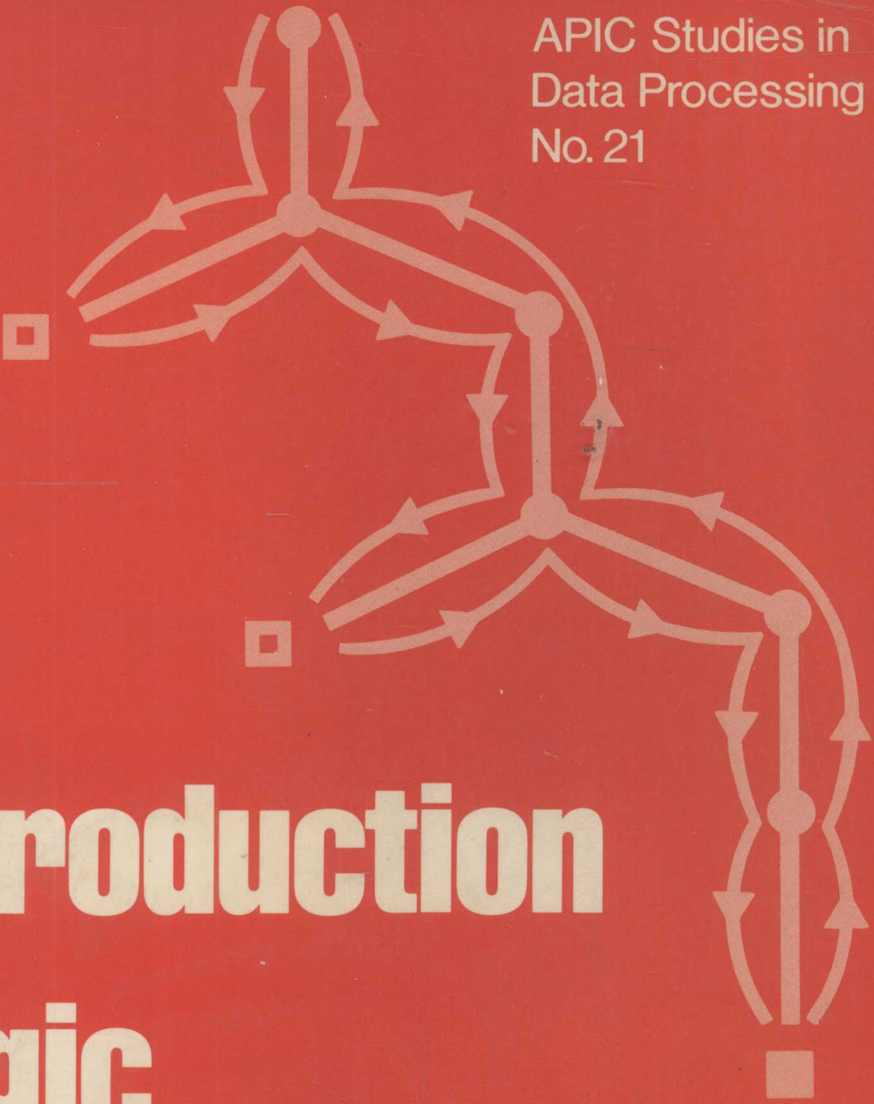
APIC Studies in Data Processing
No. 21



Introduction to Logic Programming

Christopher John Hogger

ACADEMIC PRESS, INC. (Harcourt Brace Jovanovich, Publishers)

INTRODUCTION TO LOGIC PROGRAMMING

Christopher John Hogger

Department of Computing
Imperial College of Science and Technology
London, United Kingdom

1984

ACADEMIC PRESS
(Harcourt Brace Jovanovich, Publishers)
London Orlando San Diego New York
Toronto Montreal Sydney Tokyo

ACADEMIC PRESS INC. (LONDON) LTD.
24-28 Oval Road,
London NW1 7DX

United States Edition published by
ACADEMIC PRESS, INC.
Orlando, Florida 32887

British Library Cataloguing in Publication Data

Hogger, Christopher John
Introduction to logic programming.–(APIC studies in data processing)
1. Electronic digital computers–Programming
2. Logic, Symbolic and mathematical
I. Title II. Series
001.64'2'015113 QA76.6

Library of Congress Cataloging in Publication Data

Hogger, Christopher John.
Introduction to logic programming.

Bibliography: p.
Includes index.
1. Electronic digital computers–Programming. 2. Logic, Symbolic and mathematical. I. Title.
QA76.6.H624 1984 001.64'2 84-14533
ISBN 0–12–352090–8 (alk. paper)
ISBN 0–12–352092–4 (paperback)

PRINTED IN THE UNITED STATES OF AMERICA

84 85 86 87 9 8 7 6 5 4 3 2 1

INTRODUCTION TO LOGIC PROGRAMMING

This is volume 21 in A.P.I.C. Studies in Data Processing
General Editors: Fraser Duncan and M. J. R. Shave
A complete list of titles in this series appears at the end of this volume

FOREWORD

This book is a major contribution to logic programming. It sets out for the first time in one place a comprehensive yet accessible introduction to all aspects of our subject. It covers a sound middle ground between the practical introductions to PROLOG by Clocksin and Mellish and by Clark and McCabe on the one side, and more general treatments of computational logic like Robinson's and mine on the other.

It covers two important aspects of logic programming which are to be found in no other place: derivation of logic programs from logic specifications and implementation of PROLOG. The first of these is a major contribution of logic programming to classical problems of software engineering, in which the author himself has played a significant and pioneering role. The second is a topic of great theoretical and commercial interest, and many devotees of PROLOG will be grateful for this accessible account of this hitherto esoteric subject.

This beautifully written book will be a joy to both novices and experts. It will help waken the novice to the wider world of logic programming which lies beyond PROLOG, and it will help stir the logic programming expert to greater understanding and further enthusiasm for our subject.

Imperial College, London
May 1984

ROBERT KOWALSKI

PREFACE

It is widely expected that symbolic logic will serve as the core programming formalism for the next generation of computer systems. The identification of this role for logic in Japan's Fifth Generation Project has stimulated world-wide interest in logic programming, although even beforehand it was becoming clear that this fascinating formalism was destined to make substantial contributions to the theory and practice of computing.

The advancement of logic programming was, until this recent growth of interest, confined to just a few research centres, with the result that the existing literature base describing the subject is still rather small. A few texts are already available based upon specific implementations, and others are expected shortly: for the most part these books are intended as tutorial introductions to program writing. In addition, collections of advanced research papers for computer science specialists have been published in book form. This leaves a rather large gap between the two extremes of expository level, and the chief purpose of this book is to close the gap a little.

In the book's first half, logic programming is introduced at a tutorial level but supplemented with more background and foundational material than would normally be expected in a programmer's guide. The level of presentation here is consistent with a first-year undergraduate course in computing science. The second half deals with more advanced aspects of logic as a computational formalism. It aims to gather together, simplify and interpret selected themes from a somewhat disunited, and often technically very difficult, research literature and to survey current developments in theory and application. It is intended to be of use for explanatory and reference purposes both to undergraduates taking specialized course options in logic programming and to researchers comparatively new to the field.

MEANINGS OF PRINCIPAL SYMBOLS

SYMBOL	*MEANING*
, (comma)	and
$\vee$	or
$\neg$	not
$\leftarrow$	if
$\leftrightarrow$	if and only if
iff	if and only if
$\forall$	for all
$\exists$	for some (there exists)
$:=$	is assigned
$\models$	logically implies
$\vdash$	admits a proof of
$\sim$	not provable
$\mid$	such that
$\square$	success (by contradiction)
$\blacksquare$	failure
$\neq$	not identical to
$\oplus$	vector sum
$\triangle$	assert if provable (generate lemma)
$\in$	belongs to (is a member of)
$\subseteq$	is a subset of
$\cup$	set union
$\varnothing$	empty set

CONTENTS

INTRODUCTION

The subject of this book is the use of symbolic logic as a programming language. At the time of writing, this use of logic has a history of no more than twelve years, and is still unfamiliar in detail to much of the programming community. This situation is likely to change rapidly owing to the recent identification of logic programming as the key formalism for the next generation of computers.

Logic programming differs fundamentally from conventional programming in requiring us to describe the logical structure of problems rather than making us prescribe how the computer is to go about solving them. People with no previous computing experience tend to believe that programming is, and always has been, a naturally logical business, and when introduced to languages like BASIC are often surprised or even dismayed to find that this is not really the case. Instead they discover that the traditional way of writing programs pays much homage to the computer's internal mechanisms, which, whilst certainly having a rationale of their own, do not seem to derive straightforwardly from the original conception of the problem. Conversely, programmers trained only in the use of conventional languages can experience comparable problems of adjustment when introduced to logic programming. Instinctively anxious to control the machine efficiently, they are subject to a vague sense of deprivation when getting used to a language possessing no machine-oriented features; they may suffer the programmer's equivalent of withdrawal symptoms following a long spell of addictive devotion to the assignment statement.

These adjustments are not always easy, as I know from my own experience. Two memories still stand out clearly: first, as a science undergraduate in an introductory FORTRAN course, being able to accept descriptions of the effects of individual statements upon the machine but uncertain as to how they should be knitted together in a manner consistent with the problem's logical structure; second, as a teacher of FORTRAN some years later, being shown my first logic programming statement and being unclear as to how it could contribute to an algorithmic solution of a problem on a machine. A definite effort is needed to overcome long-term conditioning to one view of computation.

The first logic programming statement I was ever shown stated "you are healthy if you eat porridge." This sentence of everyday language becomes a sentence of symbolic logic when arranged in the more structured style

healthy(*u*) **if** **eat**(*u, PORRIDGE*)

This format exposes all the principal constituents of the language as we shall use it for programming purposes: individual objects (*PORRIDGE*), variables (*u*) standing for any objects, propositions like **healthy**(*u*) and **eat**(*u, PORRIDGE*) about objects, and connectives (**if**) relating the propositions. This sentence might form part of an "expert system" program offering advice (in this example, of questionable worth) about personal nutrition. We can just as easily state something having a more numerical flavour such as

even-number(*u*) **if** **divisible**(*u, 2*)

How can logical descriptions of this sort be used to make a computer solve a problem? Consider an analogy. You wish to undertake a car journey having decided the origin, destination and possibly other defining features of the route. Getting the car to traverse this route entails multifarious decisions, many of them petty and repetitious, about vehicle-handling and motoring protocols. It is now proposed that the journey be accomplished without your having to make any of these decisions. How is this possible? By engaging someone else to do the driving and telling him the requirements of the route.

The logic programmer's 'driver' is the logic interpreter. This is a program which knows how to exploit the computer in order to infer the consequences of any set of logic sentences. The programmer's responsibility is to ensure that the given sentences are both correct and sufficiently informative to make the desired consequences inferrable.

Let's consider a more concrete example. Imagine that some piece of equipment exhibits a three-light display monitor; each light is either *ON* or *OFF*. Various *ON*/*OFF* combinations on this display periodically determine whether some switch on the equipment is to be set to *ON* or *OFF* by a human operator. Using logic we can write a collection of simple assertions

R1 : **rule**(*ON*, *ON*, *OFF*, *ON*)
R2 : **rule**(*OFF*, *ON*, *ON*, *OFF*)
⋮
etc.

where each proposition **rule**(w, x, y, z) is read as saying that the switch is to be set to the state w when the display's state is x, y, z. Jointly these sentences can be used as a decision table. Suppose we wish to store this table in a computer so that the operator can interrogate it in order to find out which state w is the appropriate response to some x, y, z displayed on the monitor. Given a logic interpreter implemented on the computer, the operator need only ask whether a particular proposition is a consequence of (implied by) the stored sentences. For example, he can type in the logic query

? **rule**(w, ON, OFF, ON)

which asks which value of w makes **rule**(w, ON, OFF, ON) such a consequence. According to R1 this value is ON and so the interpreter will autonomously discover this and print out $w := ON$.

Many other queries are answerable on the same basis (that is, without altering the stored sentences) by simply posing them to the interpreter. The query ? **rule**(OFF, x, y, z) returns all states x, y, z of the display requiring the OFF response. The query ? **rule**(OFF, ON, ON, OFF) merely asks for confirmation that OFF is a correct response to the display ON, ON, OFF; the interpreter just answers "YES" (because of R2). The query ? **rule**(w, x, y, z) elicits a printout of the entire decision table. The query ? **rule**($w1$, x, y, z), **rule**($w2$, x, y, z), $w1 \neq w2$ instigates a table-consistency check by asking whether any displays x, y, z have multiple occurrences in the table with contradictory responses. If we store in the machine two further sentences

S1 : **state**(ON)
S2 : **state**(OFF)

then the query ? **state**(x), **state**(y), **state**(z), $\sim$**rule**(w, x, y, z) instigates a table-completeness check by asking whether any displays x, y, z have been omitted from the table ($\sim$ means 'not') and, if so, tells us what they are.

In short, every possible query that is logically answerable using the stored sentences *will* be answered by the interpreter using logical inference. In virtually every other programming language each new problem to be solved using a fixed corpus of knowledge requires the laborious construction of new code, and the more complicated the derivation of the problem's solution, the more complicated that code needs to be. This inflexibility of programs in the face of changing goals must detract significantly from programming productivity.

Now it would be wrong to give the false impression that logic frees the programmer from pragmatic considerations. In realistic applications it is often necessary in the interests of acceptable execution performance to structure the input sentences with due regard for the interpreter's deductive strategy and for the particular query being posed. So the programmer will normally need to think about the algorithmic quality of what he writes as well as its descriptive quality. Nonetheless the important point is that the program statements (i.e., the input logic sentences) will always be logical descriptions of the problem itself and not of the execution process: the exact assumptions made about the problem will always be directly apparent from the program text. Throughout the book much emphasis will be placed upon this point and the ramifications it has for programming methodology and the wider issues of software engineering.

Logic programming has been successfully taught to young children using informal notions of logical implication and inference. Such informality is extremely useful for enabling naive users to assimilate the basic principles relatively painlessly. However, a more precise treatment is necessary for a proper appreciation of the formalism's historical and theoretical foundations. With this in mind, the first chapter aims to provide a reasonably precise and self-contained account of logic as a language for problem solving, explaining sentence structure, implication and inference. The second chapter has a more computational flavour, dealing mostly with the procedural interpretation of logic and showing how familiar algorithmic processes are elicited from programs by the interpreter. Chapters III and IV describe pragmatic and stylistic considerations in the structuring of programs and data. This first half of the book is therefore chiefly concerned with how to understand and construct logic programs.

The second half of the book is written more for the computer scientist and is consequently more technical. Chapters V and VI discuss the specification, verification and synthesis of programs, whilst Chapter VII outlines the elementary features of typical logic implementations. The last chapter surveys the main contributions of logic programming to computing generally: it covers important results in the theory of logic programming, describes some of the work underway in knowledge-based applications and explains the role of logic in the forthcoming fifth generation computer systems.

1 REPRESENTATION AND REASONING

A logic program consists of sentences expressing knowledge relevant to the problem that the program is intended to solve. The formulation of this knowledge makes use of two basic concepts: the existence of discrete objects, referred to here as *individuals*; and the existence of *relations* between them. The individuals considered in the context of a particular problem jointly constitute the *domain* of that problem. For example, if the problem is to solve an algebraic equation, then the domain may consist of—or at least include—the real numbers.

In order to be represented by a symbolic system such as logic, both individuals and relations must be given *names*. Naming is just a preliminary task in creating symbolic models representing what we know. The main task is to construct *sentences* expressing various logical properties of the named relations. Reasoning about some problem posed on the domain can be achieved by manipulating these sentences using logical *inference*. In a typical logic programming environment the programmer invents the sentences forming his program and the computer then performs the necessary inference to solve the problem. For this to be accomplished effectively the programmer must be sufficiently skilled both in representing knowledge and in understanding how it will be processed on the machine. In this chapter we introduce the language of *first-order logic* and show how it serves as a tool for representation and reasoning, and hence for computational problem solving.

1.1. Individuals

Individuals may be any objects at all. Examples are numbers, geometrical figures, equations and computer programs. Very often it suffices to give them simple names like

1   *2*   *ONE*   *TWO*   *CIRCLE*   *EQUATION-1*   *PROGRAM-2*

chosen from some prescribed vocabulary. These names are indivisible (or unstructured) and are conventionally called *constants*. Any number of these may, if desired, simultaneously name a particular individual. So *ONE* and *1* could both name the individual known as the first positive integer. The choice of names is arbitrary, and so the first positive integer could (perversely) be named *3* if so desired.

Sometimes it is convenient to give individuals composite (structured) names like

TWICE(*2*)   *PLUS*(*1*, *2*)

Each of these consists of an *n-tuple* prefixed by a *functor* (or function symbol). An *n*-tuple is just any ordered collection of *n* names, so (*1*, *2*) is an example of a *2*-tuple. The enclosing parentheses serve only to clarify the start and end of the *n*-tuple and can be omitted when convenient. A *2*-tuple can be called, more simply, a *pair*, whilst a *3*-tuple can be called a *triplet*. Functors like *TWICE* and *PLUS* are also chosen arbitrarily from another prescribed vocabulary. Each one can only prefix *n*-tuples for a particular value of *n*, and is then said to be an *n-place* (or *n*-ary) *functor*. So in the present context, *TWICE* is a *1*-place (or *1*-ary or 'unary') functor whilst *PLUS* is a *2*-place (or *2*-ary or 'binary') functor.

Functors enable the construction of arbitrarily elaborate names like

PLUS(*TWICE*(*2*), *PLUS*(*1*, *TWICE*(*1*)))

This name, which might be given to the seventh positive integer [because it can be viewed as $2 * 2 + (1 + 2 * 1) = 7$], indicates that individual's dependence upon two other individuals, respectively named *TWICE*(*2*) and *PLUS*(*1*, *TWICE*(*1*)). The outermost *PLUS* essentially names that dependence.

1.2. Relations

A symbol like *TWICE* has no intrinsic meaning and so does not, in its own right, correspond to our intuitive idea of 'twice'. Fundamentally that idea refers to a particular set of pairs of numbers which, for simplicity's sake, we will restrict here to the natural numbers. One way of capturing the idea is to choose names *1*, *2*, *3*, . . . etc. for the numbers and then collect them into pairs