SYMBOLIC COMPUTATION

Harvey Abramson
Veronica Dahl

# Logic Grammars

Springer-Verlag

Harvey Abramson Veronica Dahl

# Logic Grammars

With 40 Illustrations

Springer-Verlag
New York Berlin Heidelberg
London Paris Tokyo

Harvey Abramson
Department of Computer Science
University of Bristol
Queen's Building
Bristol BS8 1TR
England

Veronica Dahl
Department of Computer Science
Simon Fraser University
Burnaby, British Columbia
Canada V5A 1S6

Library of Congress Cataloging-in-Publication Data
Abramson, Harvey.
Logic grammars.
(Symbolic computation. Artificial intelligence)
Includes bibliographies.
1. Logic programming. 2. Artificial intelligence.
I. Dahl, Veronica, 1950– . II. Title. III. Series.
QA76.6.A268 1989 006.3 88-35636

Printed on acid-free paper.

© 1989 by Springer-Verlag New York Inc.
All rights reserved. This work may not be translated or copied in whole or in part without the written permission of the publisher (Springer-Verlag, 175 Fifth Avenue, New York, New York 10010, USA), except for brief excerpts in connection with reviews or scholarly analysis. Use in connection with any form of information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed is forbidden.
The use of general descriptive names, trade names, trademarks, etc. In this publication, even if the former are not especially identified, is not to be taken as a sign that such names, as understood by the Trade Marks and Merchandise Marks Act, may accordingly be used freely by anyone.

Text prepared by authors in camera-ready form.
Printed and bound by R.R. Donnelley & Sons, Harrisonburg, Virginia.
Printed in the United States of America.

9 8 7 6 5 4 3 2 1

ISBN 0-387-96961-6 Springer-Verlag New York Berlin Heidelberg
ISBN 3-540-96961-6 Springer-Verlag Berlin Heidelberg New York

To Lynn, and also to Cali who helped with the reading.

*Harvey*

To my mother, Selvita, and to Alice in Wonderland, for having provided role models with which it was possible to identify.

To Alexander and to Rob, for restoring the music in my heart.

*Veronica*

## ACKNOWLEDGMENTS

I would like to ackowledge the influence of all the people who have stimulated my orientation toward logic, linguistics, and computational linguistics:

1. The phonetician Ivar Dahl, from whom I inherited or acquired a passion for the study of language.

2. The linguists Gabriel Bès, Alfredo Hurtado, Celia Jacubowicz, Beatriz Lavandera, Ana Maria Nethol, and Luis Prieto, whom in the precariousness of the Pampas developed my interest in formal linguistics–especially Alfredo Hurtado, with whom in exile I later started a collaboration that led to my research on Static Discontinuity for Government-Binding theory.

3. Alain Colmerauer and his group at Luminy, who introduced me to the marvels of logic programming and logic grammars and for more than two years provided me with a wonderfully friendly working atmosphere.

4. The veteran logic programming community, with whose lovely people from all countries I have always felt at home.

I would also like to thank Roland Sambuc, for our joint work on the first logic programmed expert system; Michael McCord, for our joint work on coordination and on discontinuous grammars; and all the people who, as visitors or members of my research group at Simon Fraser University, have contributed in one way or another to the work described in chapter 10: Michel Boyer, Charles Brown, Sharon Hamilton, Diane Massam, Pierre Massicotte, Brenda Orser, T. Pattabhiraman, Fred Popowich and Patrick Saint-Dizier.

Finally, I thank France, for having provided in 1975 the scholarship without which this book would never have been written, and Canada, my beautiful land of adoption. And, with special warmth, my many Latin American siblings, with whom I developed the resilience to keep aiming for the impossible.

Veronica Dahl

I would like to thank the following people and institutions for their help and or inspiration:

Ray Reiter, who while he was at the University of British Columbia, helped to de-isolate the place by arranging visits of some of the leading logic programmers and thus making it possible to get involved in the field at an early stage.

Alan Robinson, for his early encouragement of work which led to HASL and eventually to my work with grammars and language implementation. His

试读结束：需要全本请在线购买： www.ertongbook.com

combination of science, humanism, and generosity is exemplary.

David Turner for the example of SASL whose implementation in logic was so easy because the language was so elegant.

Professors Mike Rogers and John Shepherdson, and all the members of the Computer Science Department at the University of Bristol for the warmth of their reception and their interest while I spent a six month's sabbatical leave there from January to June of 1987.

Seif Haridi and the Logic Programming Group of the Swedish Institute of Computer Science and the chance to spend a few weeks there in a fine working environment.

Finally, the following three great things at UBC which helped to soothe the savaged academic:

1. The Wilson Recordings Library for providing music.

2. The Asian Studies Centre and Continuing Education for providing the opportunity of beginning to learn Japanese, and for providing a window to oriental languages and cultures.

3. And most of all ... (details on request).

Harvey Abramson

## JOINT ACKNOWLEDGMENTS

We thank the students at Simon Fraser University and the University of British Columbia who helped us refine earlier versions of the material in this book. We are very grateful to Julie Johnson, G.M. Swinkels, and Bruce Weibe who spent so much time and energy in typesetting the book. Thanks to Barry J. Evans for a careful proof-reading of a near final version of the book.

We wish to thank the referees, whose pertinent comments were extemely valuable in improving the book. We also thank Springer-Verlag for its tolerance of our "fuzzy" interpretation of deadlines.

This book was completed with help to both of us from Canada's Natural Science and Engineering Research Council. V. Dahl's work described in section 3 of chapter 10 was partially supported by an S.U.R. research contract from IBM Canada.

Another such contract with H. Abramson, although not contributing material itself, did provide some support during the time of writing the book.

We wish to thank the organizing committees of the various logic programming conferences, symposia, and workshops who made it possible to collaborate in interesting places when it was too inconvenient or tedious to drive 20 miles across Vancouver.

Finally, let us state we are indebted to too many people to mention individually. Our apologies to those whose names are not explicitly stated here.

# Table of Contents

**PART 0. Introduction and Historic Overview** ........ 1

**PART I. Grammars for Formal Languages and Linguistic Research** ........ 5

Chapter 1. What are Logic Grammars ........ 5

1. Logic Grammars – Basic Features ........ 5
2. Grammar Symbols ........ 5
3. Use of Variables and Unification ........ 7
   - 3.1. Unification ........ 7
   - 3.2. Derivation Graphs ........ 9
   - 3.3. Symbol Arguments: Producers and Consumers of Structure ........ 10
4. Tests within Rules ........ 11
5. Operators ........ 12
6. Analysis and Generation ........ 13
7. A Formal Language Example ........ 14

Chapter 2. Relation of Logic Grammars to Prolog ........ 17

1. Basic Prolog Concepts ........ 17
2. Prolog as a Language Recognizer ........ 19
3. Prolog as a Structure Builder ........ 20

Bibliographic Commentary for Part I ........ 23

**PART II. Getting Started: How to Write a Simple Logic Grammar** ........ 25

Chapter 3. Step-by-Step Development of a Natural Language Analyser ........ 25

1. Propositional Statements ........ 25
2. Obtaining Representations for Propositional Statements ........ 27
3. Syntactic and Semantic Agreement ........ 30
4. Noun Phrases ........ 31
5. Negative Sentences ........ 36
6. Interrogative Clauses ........ 39

Chapter 4. Choosing Internal Representations for Natural Language ........ 43

1. Logical Form for Querying Knowledge Represented in Logic .............. 43
2. First-Order Logic Representations .............. 45
2.1. Semantic Well Formedness .............. 46
2.2. Ambiguity .............. 46
2.3. Natural Language Determiners and Modifiers .............. 46
2.3.1. Three-Branched Quantification .............. 47
2.3.2. Quantifier Scoping .............. 52
2.4. Negation .............. 53
2.5. Meaning of Plural Forms .............. 54
2.6. Representing Sets .............. 55
3. Lambda-Calculus Representations .............. 55
Chapter 5. Developing a Logic Grammar for a Formal Application .............. 61
1. An Analyzer for Logic Programs .............. 61
2. Simple Lexical Analysis .............. 64
3. Structural Representation of a Logic Program .............. 67
4. "Compiling" Logic Programs .............. 69
5. Compiling Proof Tree Generation into Rules .............. 70
Bibliographic Commentary for Part II .............. 75

**PART III. Different Types of Logic Grammars – What Each Is Useful For** .............. 77
Chapter 6. Basic Types of Logic Grammars .............. 77
1. Metamorphosis Grammars .............. 78
2. Definite Clause Grammars .............. 79
3. Extraposition Grammars .............. 81
4. Discussion .............. 82
Chapter 7. Building Structure .............. 85
1. Parse Tree Construction .............. 85
2. Meaning Representation Buildup .............. 86
3. Automating Syntactic and Semantic Structure Buildup .............. 88
Chapter 8. Modifier Structure Grammars .............. 91
1. Separation of Syntax and Semantics .............. 91
2. Quantifier Rescoping .............. 91
3. Coordination .............. 92
4. Implementation .............. 93
5. Discussion .............. 93
Chapter 9. Definite Clause Translation Grammars and their Applications .............. 95

1. Definite Clause Translation Grammars ..... 95
2. A Compiler ..... 109
2.1. Introduction ..... 109
2.2. Lexical Analysis ..... 111
2.3. Between Lexical and Syntactic Analysis ..... 113
2.4. Syntactic Analysis ..... 115
2.5. Code Generation ..... 116
2.5.1. The Target Machine ..... 117
2.5.2. Code Generated for Statements ..... 119
2.5.3. Code for Expressions ..... 122
2.6. Assembly and Allocation ..... 126
3. A Metagrammatical Extension of DCTG Notation ..... 128
4. Grammatical Data Typing ..... 135
4.1. The Natural Numbers ..... 135
4.2. Lists ..... 137
4.3. Trees ..... 139
4.4. Infix and Prefix Notation ..... 139
4.5. Comments on Grammatical Typing ..... 140
Chapter 10. Further Expressive Power – Discontinuous Grammars ..... 143
1. The Discontinuous Grammar Family ..... 143
2. Thinking in Terms of Skips – Some Examples ..... 145
2.1. Coordination ..... 145
2.2. Right Extraposition ..... 146
2.3. Interaction between Different Discontinuous Rules ..... 149
2.4. Avoiding Artifices through Straightforward Uses of Skips ..... 149
2.5. Rules with More Than One Skip ..... 150
2.6. DGs and Free Word Order Languages ..... 150
2.6.1. Totally Free Word or Constituent Order ..... 150
2.6.2. Lexically Induced Rule Format and Free Word Order ..... 151
3. Static Discontinuity Grammars and Government-Binding Theory ..... 153
3.1. Rendering Context-Free Simplicity with Type-0 Power ..... 153
3.2. Government-Binding-Oriented Constraints ..... 155
3.3. Definition ..... 157
3.4. Implementation Considerations ..... 158
3.5. Transporting the Static Discontinuity Feature into Logic Programming ..... 158
Bibliographic Commentary for Part III ..... 161

**PART IV. Other Applications** ........ 165
Chapter 11. Other Formalisms ........ 165
1. Restriction Grammars ........ 165
2. Puzzle Grammars ........ 167
3. Discussion ........ 169
Chapter 12. Bottom Up Parsing ........ 171
1. Introduction ........ 171
2. Compiling Context-Free Rules ........ 172
Bibliographic Commentary for Part IV ........ 177

**PART V. Logic Grammars and Concurrency** ........ 179
Chapter 13. Parsing with Committment ........ 179
1. Introduction ........ 179
1.1. Sample Grammar ........ 182
1.2. The One Character Lookahead Relation ........ 182
2. Compilation to Sequential Logic Programs ........ 183
3. Compilation to Concurrent Logic Program Clauses ........ 185
4. Generalized Deterministic Grammars ........ 187
5. Related Work ........ 188
6. Parallel Parsing for Natural Language Analysis ........ 189
6.1. Sample Grammar ........ 189
6.2. Parallel Parsing Method ........ 191
Bibliographic Commentary for Part V ........ 196

**Appendices** ........ 197
I. Input/Output in Logic Grammars ........ 197
1. Input of a Sentence ........ 197
II. Implementation of Logic Grammar Formalisms ........ 199
1. SYNAL: Compiling Disc. Grammars to Prolog ........ 199
2. A Short Interpreter for Static Discontinuity Grammars ........ 201
3. Implementations of DCTGs ........ 203
3.1. An Interpreter for Definite Clause Translation Grammars ........ 203
3.2. Compiling Definite Clause Translation Grammars to Prolog ........ 205
3.3. Typing DCTGs ........ 212

**LITERATURE** ........ 215
**INDEX** ........ 225

试读结束：需要全本请在线购买： www.ertongbook.com

# PART 0

# INTRODUCTION AND HISTORIC OVERVIEW

The principles of logic programming have been applied to formal and natural language processing since 1972, mainly through the Prolog language. Starting from applications such as question answering systems, many interesting problems in natural language understanding were studied with the new insight that logic is a programming tool – an insight very much in line with previous uses of logic in computational linguistics. Thus, areas such as formal representations of natural language, grammar formalisms, methods for analysis and generation, and even very specific linguistic aspects such as coordination evolved in directions typical of the context of logic and of logic programming. In the formal language area, logic grammars have been used for implementing recognizers and compilers, also with good results, particularly in the conciseness of the systems obtained.

Traditionally, logic was considered one of the most appropriate tools for representing meaning, due to its ability to deal formally with the notion of logical consequence. Representing questions and answers in logical form typically required some extensions to classical predicate calculus. In the recent past, logic was also used to represent the information to be consulted in question-answering systems. Here also, departures from first order logic were necessary, and in general, parsing knowledge, world knowledge and the meaning of questions and answers were represented and handled through quite different formalisms, resulting in the need for interfaces to link them together.

The introduction of Prolog (PROgrammation en LOGique) by Colmerauer and others made it possible to use logic throughout, minimizing interfaces from one formalism to another. World knowledge can be represented in logical form, through facts and rules of inference from which a Prolog processor can make its own deductions as needed. Extracting logical consequences amounts to hypothesizing them and letting Prolog deduce, from the facts and rules stored, whether they indeed are logical consequences, and, if they are, in which particular instances. Questioning and answering reduces to hypothesizing the question's content and letting Prolog extract instances, if any, that make the question true with respect to the world described. Those instances become answers to the question.

Parsing itself can be left to Prolog, by representing it as a deductive process – i.e., grammars can be described as facts and rules of inference, and sentence recognition reduces to hypothesizing that the sentence in question does belong to the language, and letting Prolog prove this assumption.

The Prolog equivalent of grammar symbols, moreover, are logic structures rather than simple identifiers. This means that their arguments can be used to show and build up meaning representation, to enforce syntactic and semantic agreement, etc. In other words, the very nature of Prolog facts and rules allow us to retrieve instances from parsing that can tell us more than mere recognition: logical

representation of the sentences recognized, causes of rejection, such as semantic anomaly, etc. Thus Prolog is eminently suitable for linguistic work.

Nevertheless, writing substantial grammars in Prolog that could in practice be used as parsers did require knowledge of the language (i.e., a computer specialist's mediation), and involved caring for details that belong to the nature of Prolog's mechanism rather than to considerations of linguistics or the parsing process proper.

The introduction of metamorphosis grammars by A. Colmerauer in 1975 was the first step in making Prolog a higher level grammar description tool. Although these grammars are basically a syntactic variant of Prolog, they achieve two important improvements with respect to Prolog's syntax:

1. They allow the direct writing of type-0–like rules (in the sense of Chomsky's formal grammar classification); these rules can have more than one symbol on the left hand side.
2. They hide string manipulation concerns from the user.

Prolog grammars can now be thought of as rewriting mechanisms which assemble and manipulate trees rather than as mere procedures to be described in terms of Prolog rules and facts. In present implementations, logic grammars are automatically translated into Prolog, but they still remain a distinct formalism in their own right, and they do make life easier for the non-computer specialist–e.g., for linguists.

In 1975 then, all the pieces were laid out to build a new synthesis in the design and implementation of natural language consultable knowledge systems. A. Colmerauer exemplified in a toy system about family and friendship relationships how these pieces could be put together. One of the authors, V. Dahl, developed the first sizeable applications of this new synthesis: first an expert system for configuring computer systems, together with R. Sambuc, and then a data base system with French and Spanish front ends. Both systems were written entirely in Prolog. Logic was used throughout: as a programming tool, as the means for knowledge representation, as the language for representation of meaning for queries, and (in the form of Prolog's hidden deductive process) as the parsing and data retrieval mechanism.

Not all of these uses involved the same type of logic: for natural language representation, Dahl used a three-valued set-oriented logical system; for knowledge representation, she developed some Prolog extensions such as domains, set handling primitives, etc. Yet other developments were needed to solve problems specific to Prolog, such as dealing with negation and modifying Prolog's strict left-to-right execution strategy in order to provide a more intelligent and efficient behavior.

These extensions, as well as the link between different logic formalisms used in these systems, were also hidden from the user. From an implementation point of view, the fact that logic was used throughout made the linking of different formalisms a much simpler task than in typical data base systems, resulting in a concise formulation. For instance, the three-valued logic mentioned above was

implemented through a Prolog definition of how to evaluate its expressions, which took less than a page of code.

These techniques were soon exported to other data base and expert systems consultable in other languages (English, Portugese, etc); the main feature of the application of these techniques was the striking ease with which the transposition was achieved. An English adaptation of this system was used in a key paper by F. Pereira and D. Warren–"Definite Clause Grammars for Language Analysis." This article analyzed the logic grammar approach and compared it with the Augmented Transition Network approach, concluding that the former approach was superior.

These encouraging results prompted further research: the techniques for language analysis and for modifying execution strategy were adapted into the CHAT-80 system. M. McCord systematized and perfected the notion of slots and modifiers that had been used in the earliest analyzers, achieving a more flexible strategy for determining scope; F. Pereira developed the *extraposition grammar* formalism, specifically designed to make left extraposition descriptions easier. Dahl and McCord then joined efforts to produce a metagrammatical treatment of coordination in logic grammars, and developed as a by-product a new type of them called *modifier structure grammars* (MSGs), these are essentially extraposition rules in which the semantic components are modularly separated from the syntactic ones, and for which the building of semantic and syntactic structure, as well as the treatment of quantifier scoping, of coordination, and of the interaction between the two, is automated. Further work on coordination was produced by C. Sedogbo and L. Hirschman. The notion of automatic structure buildup that resulted from Dahl and McCord's work on coordination was isolated by H. Abramson into the *definite clause translation grammar* formalism (DCTGs). In it, natural language processing power is traded for simplicity (e.g., quantifier scoping, coordination and extraposition are no longer automated), but for other applications, semantic structure buildup is usually enough. The separation between syntactic and semantic rules is also mentioned.

In 1981, V. Dahl generalized extraposition grammars into a more powerful formalism, called *discontinuous grammars*,[1] that can deal with multiple phenomena involving discontinuity: left and right extraposition, free word order, more concise descriptions, etc. Implementation issues were investigated jointly by the authors, by Dahl and McCord, and by F. Popowich. A constrained version of these grammars was investigated by Dahl and Saint-Dizier.

A more interesting subclass of the *discontinuous grammar* family was developed by Dahl and investigated within her research group for the purpose of sentence generation using Chomsky's Government and Binding theory: *static discontinuity grammars* (SDGs). In this subclass, movement phenomena can be described statically, and the power of type-0 rules coexists with the representational simplicity of context-free–like rules (i.e., trees rather than graphs can depict a sentence's

[1] The early publications use the term *gapping* instead of *discontinuous*. This name was changed in order to avoid evoking the wrong associations, since the linguistic notion of *gap* is a different one.

derivation). Hierarchical relationships, crucial to linguistic constraints in general, are thus not lost, and linguistic theories can be accommodated more readily, by expressing these constraints in terms of node domination relationships in the parse tree.

Bottom-up parsing has been investigated by A. Porto et al., Y. Matsumoto et al. and K. Uehara et al. Miyoshi and Furukawa have developed another logic programming language specifically suited for object oriented parsing. M. Filgueiras has studied the use of cooperating rewrite processes for language analysis.

The field is active and promising. This book intends both to introduce the main concepts involving language processing developments in Prolog, and to discuss the problems typically encountered and some of the alternatives for solving them.

After an in-depth presentation of the basic material, we provide a wide rather than deep coverage of many of the related topics. Some bibliographic references are mentioned within the text at places where they are directly relevant, and at the end of each part we complete the picture with other relevant bibliographic comments.

Some chapters were written by one of the authors and revised, with suggestions and comments by the other one: chapters 1 – 4, 6 – 8, 10; section 2 of chapter 11; appendix I; and sections 1 and 2 of appendix II were written by V. Dahl. Chapters 5 and 9; section 1 of chapter 11; chapters 12 and 13; and section 3 of apppendix II were written by H. Abramson. Bibliographic commentaries were written jointly.

# PART I

# GRAMMARS FOR FORMAL LANGUAGES AND LINGUISTIC RESEARCH

## Chapter 1

## What Are Logic Grammars?

### 1. Logic Grammars – Basic Features

Logic grammars can be thought of as ordinary grammars in the sense of formal language theory, in that they comprise generalized type-0 rewriting rules–rules of the form : "rewrite $\alpha$ into $\beta$," noted:

$$\alpha \rightarrow \beta$$

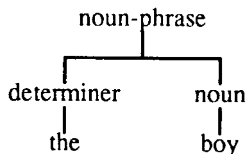
where $\alpha$ and $\beta$ are strings of terminals and nonterminals. A terminal indicates a word in the input sequence. A sequence of terminals takes the form of a Prolog list. Nonterminals indicate constituents. In this text, they take the form of a Prolog structure, where the functor names the category of the constituent and the arguments give information like number class, meaning etc.

Logic grammars differ from traditional grammars in four important respects:

1. The form of grammar symbols, which may include arguments representing trees
2. The use of variables, involving unification
3. The possibility of including tests in a rule
4. The existence of processors based on specialized theorem - provers that endow the rules with a procedural meaning by which they become parsers or synthesizers as well as descriptors for a language (e.g., Prolog and its metalevel extensions)

### 2. Grammar Symbols

Logic grammar symbols, whether terminal or nonterminal, may include arguments (as opposed to the formal grammars of Chomsky's hierarchy). One of the uses of these arguments is to construct tree structures in the course of parsing. A tree such as



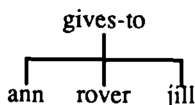
is represented in functional notation:

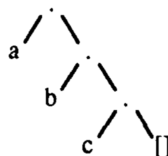
noun-phrase(determiner(the),noun(boy))

More generally, arguments have the form:

root($t_1, \cdots, t_n$)

where *root* is an $n$-ary function symbol (in our example, the binary symbol *noun–phrase*), and the $t_i$s are arguments, which in turn represent trees. Note that *argument* is used recursively. By convention (as in Prolog) arguments written in lower case are *constants*. An argument can also be a *variable*, in which case it stands for a particular but unidentified tree or constant. Variable names start with a capital. When $n=0$ (i.e., when the root has no branches), the argument is a constant (e.g. "the," "boy") or a tree consisting of just the root. Here are two more sample trees and their functional representations: (Terminal symbols are noted in square brackets.)




gives-to(ann,rover,jill) .(a,.(b,.(c,[])))

In the second of these, the root is the symbol ".", usually used to denote "concatenation." Trees constructed with this binary symbol, as above, are called *lists* and, in logic grammar are interpreted as a string of terminals. They have the simpler equivalent notation:

[a,b,c]

Summarizing, a logic grammar symbol has the form:

name($t_1, \cdots, t_n$)

where the arguments $t_i$ are either constants, variables, or trees in functional notation. Terminal symbols are enclosed in square brackets to distinguish them from nonterminal ones.

Here are some (unrelated) sample logic grammar rules:

1. verb(regarder) --> [look].
2. verb(third,singular,regarder) --> [looks].
3. a,[b] --> [b],a.

试读结束：需要全本请在线购买： www.ertongbook.com