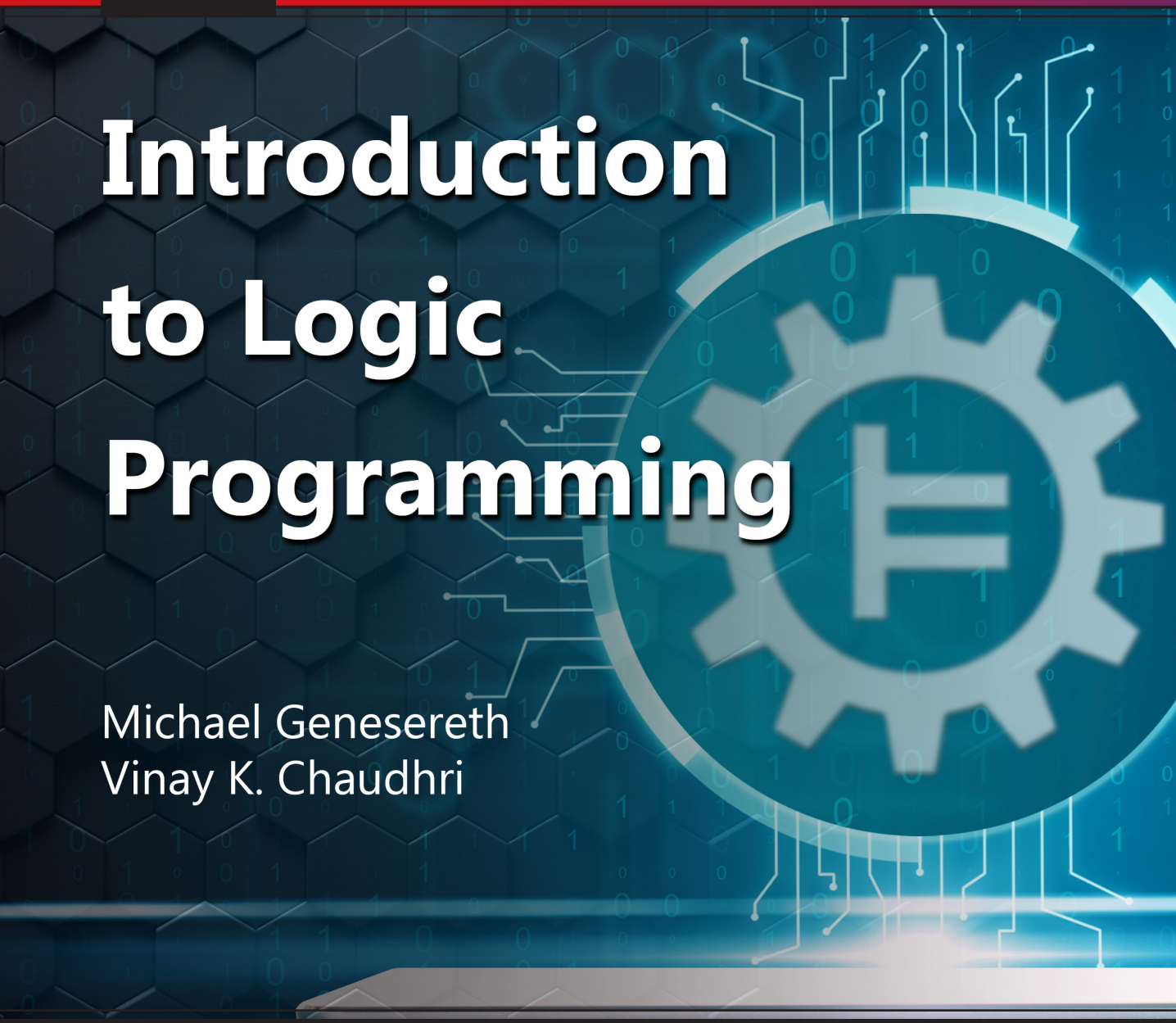
MORGAN & CLAYPOOL PUBLISHERS



Introduction to Logic Programming

Michael Genesereth
Vinay K. Chaudhri

Synthesis Lectures on Artificial Intelligence and Machine Learning

Ronald J. Brachman, Francesca Rossi, and Peter Stone, *Series Editors*

Testimonials for

Introduction to Logic Programming

This is a book for the 21st century: presenting an elegant and innovative perspective on logic programming. Unlike other texts, it takes datasets as a fundamental notion, thereby bridging the gap between programming languages and knowledge representation languages; and it treats updates on an equal footing with datasets, leading to a sound and practical treatment of action and change.

Bob Kowalski, Professor Emeritus, Imperial College London

In a world where Deep Learning and Python are the talk of the day, this book is a remarkable development. It introduces the reader to the fundamentals of traditional Logic Programming and makes clear the benefits of using the technology to create runnable specifications for complex systems.

Son Cao Tran, Professor in Computer Science, New Mexico State University

Excellent introduction to the fundamentals of Logic Programming. The book is well-written and well-structured. Concepts are explained clearly and the gradually increasing complexity of exercises makes it so that one can understand easy notions quickly before moving on to more difficult ideas.

George Younger, student, Stanford University

Introduction to Logic Programming

Synthesis Lectures on Artificial Intelligence and Machine Learning

Human Computation
Edith Law and Luis von Ahn
2011

Trading Agents
Michael P. Wellman
2011

Visual Object Recognition
Kristen Grauman and Bastian Leibe
2011

Learning with Support Vector Machines
Colin Campbell and Yiming Ying
2011

Algorithms for Reinforcement Learning
Csaba Szepesvári
2010

Data Integration: The Relational Logic Approach
Michael Genesereth
2010

Markov Logic: An Interface Layer for Artificial Intelligence
Pedro Domingos and Daniel Lowd
2009

Introduction to Semi-Supervised Learning
XiaojinZhu and Andrew B.Goldberg
2009

Action Programming Languages
Michael Thielscher
2008

Representation Discovery using Harmonic Analysis
Sridhar Mahadevan
2008

Essentials of Game Theory: A Concise Multidisciplinary Introduction
Kevin Leyton-Brown and Yoav Shoham
2008

A Concise Introduction to Multiagent Systems and Distributed Artificial Intelligence
Nikos Vlassis
2007

Intelligent Autonomous Robotics: A Robot Soccer Case Study
Peter Stone
2007

Introduction to Logic Programming

Michael Genesereth and Vinay K. Chaudhri

www.morganclaypool.com

ISBN: 9781681737225 paperback
ISBN: 9781681737232 ebook
ISBN: 9781681737249 hardcover

DOI 10.2200/S00966ED1V01Y201911AIM044

A Publication in the Morgan & Claypool Publishers series
SYNTHESIS LECTURES ON ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING

Lecture #44
Series Editors: Ronald Brachman, *Jacobs Technion-Cornell Institute at Cornell Tech*
Francesca Rossi, *IBM Research AI*
Peter Stone, *University of Texas at Austin*
Series ISSN
Synthesis Lectures on Artificial Intelligence and Machine Learning
Print 1939-4608 Electronic 1939-4616

Introduction to Logic Programming

Michael Genesereth and Vinay K. Chaudhri
Stanford University

SYNTHESIS LECTURES ON ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING #44

MORGAN & CLAYPOOL PUBLISHERS

ABSTRACT

Logic Programming is a style of programming in which programs take the form of sets of sentences in the language of Symbolic Logic. Over the years, there has been growing interest in Logic Programming due to applications in deductive databases, automated worksheets, Enterprise Management (business rules), Computational Law, and General Game Playing. This book introduces Logic Programming theory, current technology, and popular applications.

In this volume, we take an innovative, model-theoretic approach to logic programming. We begin with the fundamental notion of datasets, i.e., sets of ground atoms. Given this fundamental notion, we introduce views, i.e., virtual relations; and we define classical logic programs as sets of view definitions, written using traditional Prolog-like notation but with semantics given in terms of datasets rather than implementation. We then introduce actions, i.e., additions and deletions of ground atoms; and we define dynamic logic programs as sets of action definitions.

In addition to the printed book, there is an online version of the text with an interpreter and a compiler for the language used in the text and an integrated development environment for use in developing and deploying practical logic programs.

KEYWORDS

logic programming, computational logic, knowledge representation, deductive databases, aritificial intelligence

Contents

Preface

This book is an introductory textbook on Logic Programming. It is intended primarily for use at the undergraduate level. However, it can be used for motivated secondary school students, and it can be used at the start of graduate school for those who have not yet seen the material.

There are just two prerequisites. The book presumes that the student understands sets and set operations, such as union, intersection, and so forth. The book also presumes that the student is comfortable with symbolic mathematics, at the level of high-school algebra or beyond. Nothing else is required.

While experience in computational thinking is helpful, it is not essential. And prior programming experience is not necessary. In fact, we have observed that some students with programming backgrounds have *more* difficulty at first than students who are not accomplished programmers! It is almost as if they need to unlearn some things in order to appreciate the power and beauty of Logic Programming.

The approach to Logic Programming taken here emerged from more than 30 years of research, applications, and teaching of this material in both academic and commercial settings. The result of this experience is an approach to the subject matter that differs somewhat from the approach taken in other books on the subject in two essential ways.

First of all, in this volume, we take a model-theoretic approach to specifying semantics rather than the traditional proof-theoretic approach. We begin with the fundamental notion of *datasets*, i.e., sets of ground atoms. Given this fundamental notion, we introduce classic logic programs as *view definitions*, written using traditional Prolog notation but with semantics given in terms of datasets rather than implementation. (We also talk about implementation, but it comes later in the presentation.)

Another difference from other books on Logic Programming is that we treat change on an equal footing with state. Having talked about datasets, we introduce the fundamental notion of *updates*, i.e., additions and deletions of ground atoms. Given this fundamental notion, we introduce dynamic logic programs as sets of *action definitions*, where actions are conceptualized as sets of simultaneous updates. This extension allows us to talk about *logical agents* as well as static *logic programs*. (A logical agent is effectively a state machine in which each state is modeled as a dataset and each arc is modeled as a set of updates.)

In addition to the text of the book in print and online, there is a website with automatically graded online exercises, programming assignments, Logic Programming tools, and a variety of sample applications. The website (`http://logicprogramming.stanford.edu`) is free to use and open to all.

In conclusion, we first of all want to acknowledge the influence of two individuals who had a profound effect on our work here - Jeff Ullman and Bob Kowalski. Jeff Ullman, our colleague at Stanford, inspired us with his popular textbooks and helped us to appreciate the deep relationship between Logic Programming and databases. Bob Kowalski, co-inventor of Logic Programming, listened to our ideas, nurtured our work, and even collaborated on some of the material presented here.

We also want to acknowledge the contributions of a former graduate student - Abhijeet Mohapatra. He is a co-inventor of dynamic logic programming and the co-creator of many of the programming tools associated with our approach to Logic Programming. He helped to teach the course, worked with students, and offered invaluable suggestions on the presentation and organization of the material.

Finally, our thanks to the students who have had to endure early versions of this material, in many cases helping to get it right by suffering through experiments that were not always successful. It is a testament to the intelligence of these students that they seem to have learned the material despite multiple mistakes on our part. Their patience and constructive comments were invaluable in helping us to understand what works and what does not.

Michael Genesereth and Vinay K. Chaudhri
December 2019

PART I

Introduction

CHAPTER 1

Introduction

1.1 PROGRAMMING IN LOGIC

Logic Programming is a style of programming in which programs take the form of sets of sentences in the language of Symbolic Logic. Programs written in this style are called *logic programs*. The language in which these programs are written is called *logic programming language*. And a computer system that manages the creation and execution of logic programs is called a *logic programming system*.

1.2 LOGIC PROGRAMS AS RUNNABLE SPECIFICATIONS

Logic Programming is often said to be *declarative* or *descriptive* and contrasts with the *imperative* or *prescriptive* approach to programming associated with traditional programming languages.

In imperative/prescriptive programming, the programmer provides a detailed operational program for a system in terms of internal processing details (such as data types and variable assignments). In writing such programs, programmers typically take into account information about the intended application areas and goals of their programs, but that information is rarely recorded in the resulting programs, except in the form of non-executable comments.

In declarative/descriptive programming, programmers explicitly encode information about the application area and the goals of the program, but they do not specify internal processing details, leaving it to the systems that execute those programs to decide on those details on their own.

As an intuitive example of this distinction, consider the task of programming a robot to navigate from one point in a building to a second point. A typical imperative program would direct the robot to move forward a certain amount (or until its sensors indicated a suitable landmark); it would then tell the robot to turn and move forward again; and so forth until the robot arrives at the destination. By contrast, a typical declarative program would consist of a map and an indication of the starting and ending points on the map and would leave it to the robot to decide how to proceed.

A logic program is a type of declarative program in that it describes the application area of the program and the goals the programmer would like to achieve. It focusses on *what* is true and *what* is wanted rather than *how* to achieve the desired goals. In this respect, a logic program is more of a *specification* than an *implementation*.

Logic Programming is practical because there are well-known mechanical techniques for executing logic programs and/or producing traditional programs that achieve the same results. For this reason, logic programs are sometimes called *runnable specifications*.

1.3 ADVANTAGES OF LOGIC PROGRAMMING

Logic programs are typically *easier to create* and *easier to modify* than traditional programs. Programmers can get by with little or no knowledge of the capabilities and limitations of the systems executing those programs, and they do not need to choose specific methods of achieving their programs' goals.

Logic programs are *more composable* than traditional programs. In writing logic programs, programmers do not need to make arbitrary choices. As a result, logic programs can be combined with each other more easily than traditional programs where unnecessary arbitrary choices can conflict.

Logic programs are also more *agile* than traditional programs. A system executing a logic program can readily adapt to unexpected changes to its assumptions and/or its goals. Once again consider the robot described in the preceding section. If a robot running a logic program learns that a corridor is unexpectedly closed, it can choose a different corridor. If the robot is asked to pick up and deliver some goods along the way, it can combine routes to accomplish both tasks without having to accomplish them individually.

Finally, logic programs are more *versatile* than traditional programs—they can be used for multiple purposes, often without modification. Suppose we have a table of parents and children. Now, imagine that we are given definitions for standard kinship relations. For example, we are told that a grandparent is the parent of a parent. That single definition can be used as the basis for multiple traditional programs. (1) We can use it to build a program that computes whether one person is the grandparent of a second person. (2) We can use the definition to write a program to compute a person's grandparents. (3) We can use it to compute the grandchildren of a given person. (4) And we can use it to compute a table of grandparents and grandchildren. In traditional programming, we would write different programs for each of these tasks, and the definition of grandparent would not be *explicitly* encoded in any of these programs. In Logic Programming, the definition can be written just once, and that single definition can be used to accomplish all four tasks.

As another example of this (due to John McCarthy), consider the fact that, if two objects collide, they typically make a noise. This fact about the world can be used in designing programs for various purposes. (1) If we want to wake someone else, we can bang two objects together. (2) If we want to avoid waking someone, we would be careful *not* to let things collide. (3) If we see two cars come close in the distance and we hear a bang, we can conclude that they had collided. (4) If we see two cars come close together but we do not hear anything, we might guess that they did not collide.

1.4 APPLICATIONS OF LOGIC PROGRAMMING

Logic Programming can be used fruitfully in almost any application area. However, it has special value in application areas characterized by large numbers of definitions and constraints and rules of action, especially where those definitions and constraints and rules come from multiple sources or where they are frequently changing. The following are a few application areas where Logic Programming has proven particularly useful.

Database Systems. By conceptualizing database tables as sets of simple sentences, it is possible to use Logic in support of database systems. For example, the language of Logic can be used to define virtual views of data in terms of explicitly stored tables; it can be used to encode constraints on databases; it can be used to specify access control policies; and it can be used to write update rules.

Logical Spreadsheets/Worksheets. Logical spreadsheets (sometimes called worksheets) generalize traditional spreadsheets to include logical constraints as well as traditional arithmetic formulas. Examples of such constraints abound. For example, in scheduling applications, we might have timing constraints or restrictions on who can reserve which rooms. In the domain of travel reservations, we might have constraints on adults and infants. In academic program sheets, we might have constraints on how many courses of varying types that students must take.

Data Integration. The language of Logic can be used to relate the concepts in different vocabularies and thereby allow users to access multiple, heterogeneous data sources in an integrated fashion, giving each user the illusion of a single database encoded in his own vocabulary.

Enterprise Management. Logic Programming has special value in expressing and implementing *business rules* of various sorts. Internal business rules include enterprise policies (e.g., expense approval) and workflow (who does what and when). External business rules include the details of contracts with other enterprises, configuration and pricing rules for company products, and so forth.

Computational Law. Computational Law is the branch of Legal Informatics concerned with the representation of rule and regulations in computable form. Encoding laws in computable form enables automated legal analysis and the creation of technology to make that analysis available to citizens, and monitors and enforcers, and legal professionals.

General Game Playing. General game players are systems able to accept descriptions of arbitrary games at runtime and able to use such descriptions to play those games effectively without human intervention. In other words, they do not know the rules until the games start. Logic Programming is widely used in General Game Playing as the preferred way to formalize game descriptions.

1.5 BASIC LOGIC PROGRAMMING

Over the years, various types of Logic Programming have been explored (Basic Logic Programming, Classic Logic Programming, Transaction Logic Programming, Constraint Logic Programming, Disjunctive Logic Programming, Answer Set Programming, Inductive Logic Programming, etc.). Along with these different types of Logic Programming, a variety of logic programming languages have been developed (e.g., Datalog, Prolog, Epilog, Golog, Progol, LPS, etc.). In this volume, we concentrate on Basic Logic Programming, a variant of Transaction Logic Programming; and we use Epilog in writing our examples.

In Basic Logic Programming, we model the states of an application as sets of simple facts (called *datasets*), and we write *rules* to define abstract *views* of the facts in datasets. We model changes to state as *primitive updates* to our datasets, i.e., sets of additions and deletions of facts, and we write *rules* of a different sort to define *compound actions* in terms of primitive updates.

Epilog (the language we use in this volume) is closely related to Datalog and Prolog. Their syntaxes are almost identical. And the three languages are nicely ordered in terms of expressiveness—with Datalog being a subset of Prolog and Prolog being a subset of Epilog. For the sake of simplicity, we use the syntax of Epilog throughout this course, and we talk about the Epilog interpreter and compiler. Thus, when we mention Datalog in what follows, we are referring to the Datalog subset of Epilog; and, when we mention Prolog, we are referring to the Prolog subset of Epilog.

As we shall see, all three of these languages (Datalog and Prolog and Epilog) are less expressive than the languages associated with more complex forms of Logic Programming (such as Disjunctive Logic Programming and Answer Set Programming). While these restrictions limit what we can say in these languages, the resulting programs are computationally better behaved and, in most cases, more practical than programs written in more expressive languages. Moreover, due to these restrictions, Datalog and Prolog and Epilog are easy to understand; and, consequently, they have pedagogical value as an introduction to more complex Logic Programming languages.

In keeping with our emphasis on Basic Logic Programming, the material of the course is divided into five units. In this unit, Unit 1, we give an overview of Logic Programming and Basic Logic Programming, and we introduce *datasets*. In Unit 2, we talk about *queries* and *updates*. In Unit 3, we talk about *view definitions*. In Unit 4, we concentrate on *operation definitions*. And, in Unit 5, we talk about *variations*, i.e., other forms of Logic Programming.

HISTORICAL NOTES

In the mid-1950s, computer scientists began to concentrate on the development of high-level programming languages. As a contribution to this effort, John McCarthy suggested the language of Symbolic Logic as a candidate, and he articulated the ideal of declarative programming. He

gave voice to these ideas in a seminal paper, published in 1958, which describes a type of system that he called an *advice taker*.

> "The main advantage we expect the advice taker to have is that its behavior will be improvable merely by making statements to it, telling it about its ... environment and what is wanted from it. To make these statements will require little, if any, knowledge of the program or the previous knowledge of the advice taker."

The idea of declarative programming caught the imaginations of subsequent researchers—notably Bob Kowalski, one of the fathers of Logic Programming, and Ed Feigenbaum, the inventor of Knowledge Engineering. In a paper written in 1974, Feigenbaum gave a forceful restatement of McCarthy's ideal.

> "The potential use of computers by people to accomplish tasks can be 'one-dimensionalized' into a spectrum representing the nature of the instruction that must be given the computer to do its job. Call it the what-to-how spectrum. At one extreme of the spectrum, the user supplies his intelligence to instruct the machine with precision exactly how to do his job step-by-step. ... At the other end of the spectrum is the user with his real problem. ... He aspires to communicate what he wants done ... without having to lay out in detail all necessary subgoals for adequate performance."

The development of Logic Programming in its present form can be traced to subsequent debates about declarative vs. procedural representations of knowledge in the Artificial Intelligence community.

Advocates of procedural representations were mainly centered at MIT, under the leadership of Marvin Minsky and Seymour Papert. Although it was based on the proof methods of logic, Planner, developed at MIT, was the first language to emerge within the proceduralist paradigm. Planner featured pattern-directed invocation of procedural plans from goals (i.e., goal-reduction or backward chaining) and from assertions (i.e., forward chaining). The most influential implementation of Planner was the subset of Planner, called Micro-Planner, implemented by Gerry Sussman, Eugene Charniak and Terry Winograd. It was used to implement Winograd's natural-language understanding program SHRDLU, which was a landmark at that time.

Advocates of declarative representations were centered at Stanford (associated with John McCarthy, Bertram Raphael, and Cordell Green) and in Edinburgh (associated with John Alan Robinson, Pat Hayes, and Robert Kowalski). Hayes and Kowalski tried to reconcile the logic-based declarative approach to knowledge representation with Planner's procedural approach. In 1973, Hayes developed an equational language, Golux, in which different procedures could be obtained by altering the behavior of a theorem prover. Kowalski, on the other hand, developed SLD resolution, a variant of SL-resolution, and showed how it treats implications as goal-reduction procedures. Kowalski collaborated with Colmerauer in Marseille, who developed these ideas in the design of the programming language Prolog, which was implemented in the

summer and autumn of 1972. The first Prolog program, also written in 1972 and implemented in Marseille, was a French question-answering system. The use of Prolog as a practical programming language was given great momentum by the development of a compiler by David Warren in Edinburgh in 1977.

CHAPTER 2

Datasets

2.1 INTRODUCTION

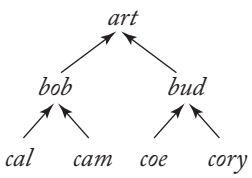
Datasets are collections of facts about some aspect of the world. Datasets can be used by themselves to encode information. They can also be used in combination with logic programs to form more complex information systems, as we shall see in the coming chapters.

We begin this chapter by talking about conceptualizing the world. We then introduce a formal language for encoding information about our conceptualization in the form of datasets. We provide some examples of datasets encoded within this language. And, finally, we discuss the issues involved in reconceptualizing an application area and encoding those different conceptualizations as datasets with different vocabularies.

2.2 CONCEPTUALIZATION

When we think about the world, we usually think in terms of objects and relationships among these objects. *Objects* include things like people and offices and buildings. *Relationships* include things like parenthood, friendship, office assignments, office locations, and so forth.

One way to represent such information is in the form of graphs. As an example, consider the graph shown below. The nodes here represent objects, and the arcs represent relationships among these objects.



Alternatively, we can represent such information in the form of tables. For example, we can encode the information in the preceding graph as a table like the one shown below.

parent	
art	bob
art	bea
bob	cal
bob	cam
bea	coe
bea	cory

Another possibility is to encode individual relationships as sentences in a formal language. For example, we can represent our kinship information as shown below. Here, each fact takes the form of a sentence consisting of name for the relationship and the names of the entities involved.

```
parent(art,bob)
parent(art,bea)
parent(bob,cal)
parent(bob,cam)
parent(bea,coe)
parent(bea,cory)
```

While graphs and tables are intuitively appealing, a sentential representation is more useful for our purposes. So, in what follows we represent facts as sentences, and we represent different states of the world as different sets of such sentences.

A final note before we leave this discussion of conceptualization. In what follows, we use the words *relation* and *relationship* interchangeably. From a mathematical point of view, this is not exactly correct, as there is a subtle difference between the two notions. However, for our purposes, the difference is unimportant, and it is often easier to say *relation* than *relationship*.

2.3 DATASETS

A *dataset* is a collection of simple facts that characterize the state of an application area. Facts in a dataset are assumed to be true; facts that are not included in the dataset are assumed to be false. Different datasets characterize different states.

Constants are strings of lower case letters, digits, underscores, and periods *or* strings of arbitrary ASCII characters enclosed by double quotes. For reasons described in the next chapter, we prohibit strings containing uppercase letters except within double quotes. Examples of constants include `a`, `b`, `comp225`, `123`, `3.14159`, `barack_obama`, and `"Mind your p's and q's!"`. Non-examples include `Art`, `p&q`, `the-house-that-jack-built`. The first contains an upper

case letter; the second contains an ampersand; and the third contains hyphens. A *vocabulary* is a collection of constants.

In what follows, we distinguish three types of constants. *Symbols* are intended to represent objects in the world. *Constructors* are used to create compound names for objects. *Predicates* represent relationships on objects.

Each constructor and predicate has an associated *arity*, i.e., the number of arguments allowed in any expression involving the constructor or predicate. *Unary* constructors and predicates are those that take one argument; *binary* constructors and predicates take two arguments; and *ternary* constructors and predicates take three arguments. Beyond that, we often say that constructors and predicates are *n-ary*. Note that it is possible to have a predicate with no arguments, representing a condition that is simply true or false.

A *ground term* is either a symbol or a compound name. A *compound name* is an expression formed from an *n*-ary constructor and *n* ground terms enclosed in parentheses and separated by commas. If `a` and `b` are symbols and `pair` is a binary constructor, then `pair(a,a)`, `pair(a,b)`, `pair(b,a)`, and `pair(b,b)` are compound names. The adjective *ground* here means that the term does not contain any *variables* (which we discuss in the next chapter).

The *Herbrand universe* for a vocabulary is the set of all ground terms that can be formed from the symbols and constructors in the vocabulary. For a finite vocabulary without constructors, the Herbrand universe is finite (i.e., just the symbols). For a finite vocabulary *with* constructors, the Herbrand universe is infinite (i.e., the symbols and all compound names that can be formed from those symbols). The Herbrand universe for the vocabulary described in the previous paragraph is shown below.

```
{pair(a,b), pair(a,pair(b,c)), pair(a,pair(b,pair(c,d))), ...}
```

A *datum/factoid/fact* is an expression formed from an *n*-ary predicate and *n* ground terms enclosed in parentheses and separated by commas. For example, if `r` is a binary predicate and `a` and `b` are symbols, then `r(a,b)` is a datum.

The *Herbrand base* for a vocabulary is the set of all factoids that can be formed from the constants in the vocabulary. For example, for a vocabulary with just two symbols `a` and `b` and the single binary predicate `r`, the Herbrand base for this language is shown below.

```
{r(a,a), r(a,b), r(b,a), r(b,b)}
```

Finally, we define a *dataset* to be any subset of the Herbrand base, i.e., an arbitrary set of facts that can be formed from the vocabulary of a database. Intuitively, we can think of the data in a dataset as the facts that we believe to be true; data that are not in the dataset are assumed to be false.

2.4 EXAMPLE – SORORITY WORLD

Consider the interpersonal relations of a small sorority. There are just four members—Abby, Bess, Cody, and Dana. Some of the girls like each other, but some do not.

Figure 2.1 shows one set of possibilities. The checkmark in the first row here means that Abby likes Cody, while the absence of a checkmark means that Abby does not like the other girls (including herself). Bess likes Cody too. Cody likes everyone but herself. And Dana also likes the popular Cody.

	Abby	Bess	Cody	Dana
Abby			✓	
Bess			✓	
Cody	✓	✓		✓
Dana			✓	

Figure 2.1: One state of Sorority World.

In order to encode this information as a dataset, we adopt a vocabulary with four symbols (`abby`, `bess`, `cody`, `dana`) and one binary predicate (`likes`). Using this vocabulary, we can encode the information in Figure 2.1 by writing the dataset shown below.

```
likes(abby,cody)
likes(bess,cody)
likes(cody,abby)
likes(cody,bess)
likes(cody,dana)
likes(dana,cody)
```

Note that the `likes` relation has no inherent restrictions. It is possible for one person to like a second without the second person liking the first. It is possible for a person to like just one other person or many people or nobody. It is possible that everyone likes everyone or no one likes anyone.

Even for a small world like this one, there are quite a few possible ways the world could be. Given four girls, there are sixteen *possible* instances of the `likes` relation—`likes(abby,abby)`, `likes(abby,bess)`, `likes(abby,cody)`, `likes(abby,dana)`, `likes(bess,abby)`, and so forth. Each of these sixteen can be either true or false. There are 2^{16} (i.e., 65,536) possible combinations of these true-false possibilities; and so there are 2^{16} possible states of this world and, therefore, 2^{16} possible datasets.

2.5 EXAMPLE – KINSHIP

As another example, consider a small dataset about kinship. The terms in this case once again represent people. The predicates name properties of these people and their relationships with each other.

In our example, we use the binary predicate `parent` to specify that one person is a parent of another. The sentences below constitute a dataset describing six instances of the `parent` relation. The person named `art` is a parent of the person named `bob` and the person named `bea`; `bob` is the parent of `cal` and `cam`; and `bea` is the parent of `coe` and `cory`.

```
parent(art,bob)
parent(art,bea)
parent(bob,cal)
parent(bob,cam)
parent(bea,coe)
parent(bea,cory)
```

The `adult` relation is a unary relation, i.e., a simple property of a person, not a relationship with other people. In the dataset below, everyone is an adult except for Art's grandchildren.

```
adult(art)
adult(bob)
adult(bea)
```

We can express gender with two unary predicates `male` and `female`. The following data expresses the genders of all of the people in our dataset. Note that, in principle, we need only one relation here, since one gender is the complement of the other. However, representing both allows us to enumerate instances of both gender equally efficiently, which can be useful in certain applications.

```
male(art)          female(bea)
male(bob)          female(coe)
male(cal)          female(cory)
male(cam)
```

As an example of a ternary relation, consider the data shown below. Here, we use `prefers` to represent the fact that the first person likes the second person more than the third person. For example, the first sentence says that Art prefers `bea` to `bob`; the second sentence says that `bob` prefers `cal` to `cam`.

```
prefers(art,bea,bob)
prefers(bob,cal,cam)
```

Note that the order of arguments in such sentences is arbitrary. Given the meaning of the `prefers` relation in our example, the first argument denotes the subject, the second argument is the person who is preferred, and the third argument denotes the person who is less preferred. We could equally well have interpreted the arguments in other orders. The important thing is consistency—once we choose to interpret the arguments in one way, we must stick to that interpretation everywhere.

One noteworthy difference difference between Sorority World and Kinship is that there is just one relation in the former (i.e., the `likes` relation), whereas there are multiple relations in the latter (three unary predicates, one binary predicate, and one ternary predicate).

A more subtle and interesting difference is that the relations in Kinship are constrained in various ways while the `likes` relation in Sorority World is not. It is possible for any person in Sorority World to like any other person; all combinations of likes and dislikes are possible. By contrast, in Kinship there are constraints that limit the number of possible states. For example, it is not possible for a person to be his own parent, and it is not possible for a person to be both male and female.

2.6 EXAMPLE – BLOCKS WORLD

The Blocks World is a popular application area for illustrating ideas in the field of Artificial Intelligence. A typical Blocks World scene is shown in Figure 2.2.

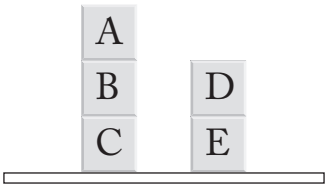


Figure 2.2: One state of Blocks World.

Most people looking at Figure 2.2 interpret it as a configuration of five toy blocks. Some people conceptualize the table on which the blocks are resting as an object as well; but, for simplicity, we ignore it here.

In order to describe this scene, we adopt a vocabulary with five symbols (`a`, `b`, `c`, `d`, `e`), with one symbol for each of the five blocks in the scene. The intent here is for each of these symbols to represent the block marked with the corresponding capital letter in the scene.

In a spatial conceptualization of the Blocks World, there are numerous meaningful relations. For example, it makes sense to talk about the relation that holds between two blocks if and only if one is resting on the other. In what follows, we use the predicate `on` to refer to this

relation. We might also talk about the relation that holds between two blocks if and only if one is anywhere above the other, i.e., the first is resting on the second or is resting on a block that is resting on the second, and so forth. In what follows, we use the predicate `above` to talk about this relation. There is the relation that holds of three blocks that are stacked one on top of the other. We use the predicate `stack` as a name for this relation. We use the predicate `clear` to denote the relation that holds of a block if and only if there is no block on top of it. We use the predicate `table` to denote the relation that holds of a block if and only if that block is resting on the table.

The arities of these predicates are determined by their intended use. Since `on` is intended to denote a relation between two blocks, it has arity 2. Similarly, `above` has arity 2. The `stack` predicate has arity 3. Predicates `clear` and `table` each have arity 1.

Given this vocabulary, we can describe the scene in Figure 2.2 by writing sentences that state which relations hold of which objects or groups of objects. Let's start with `on`. The following sentences tell us directly for each ground relational sentence whether it is true or false.

```
on(a,b)
on(b,c)
on(d,e)
```

There are four `above` facts. The `above` relation holds of the same pairs of blocks as the `on` relation, but it includes one additional fact for block `a` and block `c`.

```
above(a,b)
above(b,c)
above(a,c)
above(d,e)
```

In similar fashion, we can encode the stack relation and the above relation. There is just one stack here—block `a` on block `b` and block `b` on block `c`.

```
stack(a,b,c)
```

Finally, we can write out the facts for `clear` and `table`. Blocks `a` and `d` are clear, while blocks `c` and `e` are on the table.

```
clear(a)          table(c)
clear(d)          table(e)
```

As with Kinship, the relations in Blocks World are constrained in various ways. For example, it is not possible for a block to be on itself. Moreover, some of these relations are entirely

determined by others. For example, given the on relation, the facts about all of the other relations are entirely determined. In a later chapter, we see how to write out definitions for such concepts and thereby avoid having to write out individual facts for such defined concepts.

2.7 EXAMPLE – FOOD WORLD

As another example of these concepts, consider a small dataset about food and menus. The goal here is to create a dataset that lists meals that are available at a restaurant on different days of the week.

The symbols in this case come in two types - days of the week (`monday`, ... , `friday`) and different types of food (`calamari`, `vichyssoise`, `beef`, and so forth). There are three constructors—a 3-ary constructor for three course meals (`three`), a 4-ary constructor for four course meals (`four`), and a 5-ary constructor for five course meals (`five`). There is a single binary predicate `menu` that relates days of the week and available meals.

The following is an example of a dataset using this vocabulary. On Monday, the restaurant offers a three course meal with calamari and beef and shortcake, and it offers a different three course meal with puree and beef and ice cream for dessert. On Tuesday, the restaurant offers one of the same three-course meals and a four-course meal as well. On Wednesday, the restaurant offers just one meal—the four-course meal from the day before. On Thursday, the restaurant offers a five-course meal; and, on Friday, it offers a different five-course meal.

```
menu(monday,three(calamari,beef,shortcake))
menu(monday,three(puree,beef,icecream))
menu(tuesday,three(puree,beef,icecream))
menu(tuesday,four(consomme,greek,lamb,baklava))
menu(wednesday,four(consomme,greek,lamb,baklava))
menu(thursday,five(vichyssoise,caesar,trout,chicken,tiramisu))
menu(friday,five(vichyssoise,green,trout,beef,souffle))
```

Note that, although there are constructors here, the dataset is finite in size. In fact, there are strong restrictions on what sentences make sense. For example, only symbols representing days of the week appear as the first argument of the `menu` relation. Only symbols representing foods appear as arguments in compound names. And only whole meals appear as the second argument of the `menu` relation. Note also that compound names are not nested here. These kinds of restrictions are common in datasets. Later in the book, we show how we can formalize these constraints.

2.8 REFORMULATION

No matter how we choose to conceptualize the world, it is important to realize that there are other conceptualizations as well. Furthermore, there need not be any correspondence between

the objects, functions, and relations in one conceptualization and the objects, functions, and relations in another.

In some cases, changing one's conceptualization of the world can make it impossible to express certain kinds of knowledge. A famous example of this is the controversy in the field of physics between the view of light as a wave phenomenon and the view of light in terms of particles. Each conceptualization allowed physicists to explain different aspects of the behavior of light, but neither alone sufficed. Not until the two views were merged in modern quantum physics were the discrepancies resolved.

In other cases, changing one's conceptualization can make it more difficult to express knowledge, without necessarily making it impossible. A good example of this, once again in the field of physics, is changing one's frame of reference. Given Aristotle's geocentric view of the universe, astronomers had great difficulty explaining the motions of the moon and other planets. The data were explained (with epicycles, etc.) in the Aristotelian conceptualization, although the explanation was extremely cumbersome. The switch to a heliocentric view quickly led to a more perspicuous theory.

This raises the question of what makes one conceptualization more appropriate than another. Currently, there is no comprehensive answer to this question. However, there are a few issues that are especially noteworthy.

One such issue is the *grain size* of the objects associated with aconceptualization. Choosing too small a grain can make knowledge formalization prohibitively tedious. Choosing too large a grain can make it impossible.

As an example of the former problem, consider a conceptualization of the scene in Blocks World in which the objects in the universe of discourse are the atoms composing the blocks in the picture. Each block is composed of enormously many atoms, so the universe of discourse is extremely large. Although it is, in principle, possible to describe the scene at this level of detail, it is senseless if we are interested in only the vertical relationship of the blocks made up of those atoms. Of course, for a chemist interested in the composition of blocks, the atomic view of the scene might be more appropriate, and our conceptualization in terms of blocks has too large a grain.

Indistinguishability abstraction is a form of object reformulation that deals with grain size. If several objects mentioned in a dataset satisfy all of the same conditions, under appropriate circumstances, it is possible to abstract the objects to a single object that does not distinguish the identities of the individuals. This can decrease the cost of processing queries by avoiding redundant computation in which the only difference is the identities of these objects.

Another way of reconceptualizing the world is the *reification* of relations as objects in the universe of discourse. The advantage of this is that it allows us to consider properties of properties.

As an example, consider a Blocks World conceptualization in which there are five blocks, no constructors, and three unary predicates, each corresponding to a different color. This conceptualization allows us to consider the colors of blocks but not the properties of those colors.

We can remedy this deficiency by *reifying* various color relations as objects in their own right and by adding a relation to associate blocks with colors. Because the colors are objects in the universe of discourse, we can then add relations that characterize them, e.g., warm, cool, and so forth.

There is also the reverse of reification, viz. *relationalization*. Combining relationalization and reification is a common way to change from one conceptualization to another.

Note that, in this discussion, no attention has been paid to the question of whether the objects in one's conceptualization of the world really exist. We have adopted neither the standpoint of *realism*, which posits that the objects in one's conceptualization really exist, nor that of *nominalism*, which holds that one's concepts have no necessary external existence. Conceptualizations are our inventions, and their justification is based solely on their utility. This lack of commitment indicates the essential ontological promiscuity of Logic Programming: any conceptualization of the world is accommodated, and we seek those that are useful for our purposes.

2.9 EXERCISES

2.1. Consider the Sorority World introduced above. Write out a dataset describing a state in which every girl likes herself and no one else.

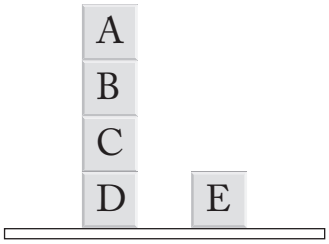
2.2. Consider a variation of the Sorority World example in which we have a single binary relation, called `friend`. `friend` differs from `likes` in two ways. It is non-reflexive, i.e., a girl cannot be friends with herself; and it is symmetric, i.e., if one girl is a friend of a second girl, then the second girl is friends with the first. Write out a dataset describing a state that satisfies the non-reflexivity and symmetry of the `friend` relation *and* so that exactly six `friend` facts are true. Note that there are multiple ways in which this can be done.

2.3. Consider a variation of the Sorority World example in which we have a single binary relation, called `younger`. `younger` differs from `likes` in three ways. It is non-reflexive, i.e., a girl cannot be younger than herself. It is antisymmetric, i.e., if one girl is younger than a second, then the second is *not* younger than the first. It is transitive, i.e., if one girl is younger than a second and the second is younger than a third, then the first is younger than the third. Write out a dataset describing a state that satisfies the reflexivity, antisymmetry, and transitivity of the younger relation *and* so that the maximum number of `younger` facts are true. Note that there are multiple ways in which this can be done.

2.4. A person x is a *sibling* of a person y if and only if x is a brother or a sister of y. Write out the `sibling` facts corresponding to the `parent` facts shown below.

```
parent (art,bob)
parent (art,bob)
parent (art,bob)
parent (art,bob)
parent (art,bob)
parent (art,bob)
```

2.5. Consider the state of the Blocks World pictured below. Write out all of the above facts that are true in this state.



2.6. Consider a world with n symbols and a single binary predicate. How many distinct facts can be written in this language?

$$n,\ 2n,\ n^2,\ 2^n,\ n^n,\ 2^{n^2},\ 2^{2^n}$$

2.7. Consider a world with n symbols and a single binary predicate. How many distinct datasets are possible for this language?

$$n,\ 2n,\ n^2,\ 2^n,\ n^n,\ 2^{n^2},\ 2^{2^n}$$

2.8. Consider a world with n symbols and a single binary predicate; and suppose that the binary relation is functional, i.e., every symbol in the first position is paired with exactly one symbol in the second position. How many distinct datasets satisfy this restriction?

$$n,\ 2n,\ n^2,\ n^n,\ 2^n,\ 2^{n^2},\ 2^{2^n}$$

PART II

Queries and Updates

CHAPTER 3

Queries

3.1 INTRODUCTION

In Chapter 2, we saw how to represent the state of an application area as a dataset. If a dataset is large, it can be difficult to answer questions based on that dataset. In this chapter, we look at various ways of *querying* a dataset to find just the information that we need.

The simplest form of query is a *true-or-false* question. Given a factoid and a dataset, we might want to know whether or not the factoid is true in that dataset. For example, we might want to know whether a person Art is the parent of Bob. Answering an atomic true-or-false question is simply a matter of checking whether the given factoid is a member of the dataset.

A more interesting form of query is a *fill-in-the-blanks* question. Given a factoid with blanks, we might want values that, when substituted for the blanks, make the query true. For example, we might want to look up the children of Art or the parents of Bill or pairs of parents and children.

An even more interesting form of query is a *compound* question. We might want values for which a Boolean combination of conditions is true. For example, we might want whether Art is the parent of Bob *or* the parent of Bud. Or we might want to find all people who have sons *and* who have *no* daughters.

We begin this chapter by looking at an extension of our dataset language that allows us to express such questions. In the next section, we define the syntax of our language; and, in the section thereafter, we define its semantics. We then look at some examples of using this language to query datasets. With that introduction behind us, we look at an important syntactic restriction, called safety. And, finally, we finish by discussing useful predefined concepts (e.g., arithmetic operators) that increase the power of our query language.

3.2 QUERY SYNTAX

The language of queries includes the language of datasets but provides some additional features that make it more expressive, viz. variables and query rules. Variables allow us to write fill-in-the-blanks queries. Query rules allow us to express compound queries, notably negations (to say that a condition is false), conjunctions (to say that several conditions are all true), and disjunctions (to say that at least one of several conditions is true).

In our query language, a *variable* is either a lone underscore or a string of letters, digits, and underscores beginning with an uppercase letter. For example, `_`, `X23`, `X_23`, and `Somebody` are all variables.

An *atomic sentence*, or *atom*, is analogous to a factoid in a dataset except that the arguments may include variables as well as symbols. For example, if `p` is a binary predicate and `a` is a symbol and `Y` is a variable, then `p(a,Y)` is an atomic sentence.

A *literal* is either an atom or a negation of an atom. A simple atom is called a *positive* literal. The negation of an atom is called a *negative* literal. In what follows, we write negative literals using the negation sign `~`. For example, if `p(a,b)` is an atom, then `~p(a,b)` denotes the negation of this atom. Both are literals.

A *query rule* is an expression consisting of a distinguished atom, called the *head* and a collection of zero or more literals, called the *body*. The literals in the body are called *subgoals*. The predicate in the head of a query rule must be a new predicate (i.e., not one in the vocabulary of our dataset), and all of the predicates in the body must be dataset predicates.

In what follows, we write rules as in the example shown below. Here, `goal(a,b)` is the head; `p(a,b) & ~q(b)` is the body; and `p(a,b)` and `~q(b)` are subgoals.

```
goal(a,b) :- p(a,b) & ~q(b)
```

As we shall see in the next section, a query rule is something like a reverse implication—it is a statement that the head of the rule (i.e., the overall goal) is true whenever the subgoals are true. For example, the rule above states that `goal(a,b)` is true *if* `p(a,b)` is true *and* `q(b)` is *not* true.

The expressive power of query rules is greatly enhanced through the use of variables. Consider, for example, the rule shown below. This is a more general version of the rule shown above. Instead of applying to just the specific objects `a` and `b` it applies to *all* objects. In this case, the rule states that `goal` is true of *any* object `X` and *any* object `Y` if `p` is true of `X` and `Y` and `q` is not true of `Y`.

```
goal(X,Y) :- p(X,Y) & ~q(Y)
```

A *query* is a non-empty, finite set of query rules. Typically, a query consists of just one rule. In fact, most Logic Programming systems do not support queries with multiple rules (at least not directly). However, queries with multiple rules are sometimes useful and do not add any major complexity, so in what follows we allow for the possibility of queries with multiple rules.

3.3 QUERY SEMANTICS

An *instance* of an expression (atom, literal, or rule) is one in which all variables have been consistently replaced by ground terms (i.e., terms without variables). For example, if we have a language with symbols `a` and `b`, then the instances of `goal(X,Y) :- p(X,Y) & ~q(Y)` are shown below.

```
goal(a,a) :- p(a,a) & ~q(a)
goal(a,b) :- p(a,b) & ~q(b)
goal(b,a) :- p(b,a) & ~q(a)
goal(b,b) :- p(b,b) & ~q(b)
```

Given this notion, we can define the result of the application of a single rule to a dataset. Given a rule r and a dataset Δ, we define $v(r,\Delta)$ to be the set of all ψ such that (1) ψ is the head of an arbitrary instance of r, (2) every positive subgoal in the instance is a member of Δ, and (3) no negative subgoal in the instance is a member of Δ.

The *extension* of a query is the set of all facts that can be "deduced" on the basis of the rules in the program, i.e., it is the union of $v(r_i, \Delta)$ for each r_i in our query.

To illustrate these definitions, consider a dataset describing a small directed graph. In the sentences below, we use symbols to designate the nodes of the graph, and we use the `p` relation to designate the arcs of the graph.

```
p(a,b)
p(b,c)
p(c,b)
```

Now suppose we were given the following query. Here, the predicate `goal` is defined to be true of every node that has an outgoing arc to another node and also an incoming arc from that node.

```
goal(X) :- p(X,Y) & p(Y,X)
```

Since there are two variables here and three symbols, there are nine instances of this rule, viz. the ones shown below.

```
goal(a) :- p(a,a) & p(a,a)
goal(a) :- p(a,b) & p(b,a)
goal(a) :- p(a,c) & p(c,a)
goal(b) :- p(b,a) & p(a,b)
goal(b) :- p(b,b) & p(b,b)
goal(b) :- p(b,c) & p(c,b)
```

```
goal(c) :- p(c,a) & p(a,c)
goal(c) :- p(c,b) & p(b,c)
goal(c) :- p(c,c) & p(c,c)
```

The body in the first of these instances is not satisfied. In fact, the body is true only in the sixth and eighth instances. Consequently, the extension of this query contains just the two atoms shown below.

```
goal(b)
goal(c)
```

The definition of semantics in terms of rule instances is simple and clear. However, Logic Programming systems typically do not implement query processing in this way. There are more efficient ways of computing such extensions. In subsequent chapters, we look at some algorithms of this sort.

3.4 SAFETY

A query rule is *safe* if and only if every variable that appears in the head or in any negative literal in the body also appears in at least one positive literal in the body.

The rule shown below is safe. Every variable in the head and every variable in the negative subgoal appears in a positive subgoal in the body. Note that it is okay for the body to contain variables that do not appear in the head.

```
goal(X) :- p(X,Y,Z) & ~q(X,Z)
```

By contrast, the two rules shown below are not safe. The first rule is not safe because the variable `Z` appears in the head but does not appear in any positive subgoal. The second rule is not safe because the variable `Z` appears in a negative subgoal but not in any positive subgoal.

```
goal(X,Y,Z) :- p(X,Y)
goal(X,Y,X) :- p(X,Y) & ~q(Y,Z)
```

To see why safety matters in the case of the first rule, suppose we had a database in which `p(a,b)` is true. Then, the body of the first rule is satisfied if we let `X` be `a` and `Y` be `b`. In this case, we can conclude that every corresponding instance of the head is true. But what should we substitute for `Z`? Intuitively, we could put anything there; but there could be many possibilities. While this is conceptually okay, it is practically problematic.

To see why safety matters in the second rule, suppose we had a database with just two facts, viz. `p(a,b)` and `q(b,c)`. In this case, if we let `X` be `a` and `Y` be `b` and `Z` be anything other than `c`, then both subgoals are true, and we can conclude `goal(a,b,a)`.

The main problem with this is that many people incorrectly interpret that negation as meaning there is no Z for which q(Y,Z) is true, whereas the correct reading is that q(Y,Z) needs to be false for just one value of Z. As we will see, there are various ways of expressing this second meaning without writing unsafe queries.

3.5 PREDEFINED CONCEPTS

In practical logic programming languages, it is common to predefine useful concepts. These typically include arithmetic functions (such as plus, times, max, min), string functions (such as concatenation), equality and inequality, aggregates (such as countofall), and so forth.

In Epilog, equality and inequality are expressed using the relations same and distinct. The sentence same(σ,τ) is true iff σ and τ are identical. The sentence distinct(σ,τ) is true if and only if σ and τ are different.

The evaluate relation is used to represent equations involving predefined functions. For example, we would write evaluate(plus(times(3,3),times(2,3),1),16) to represent the equation 3^2+2x3+1=16. If height is a binary predicate relating a figure and its height and if width is a binary predicate relating a figure and its width, we can define the area of the object as shown below. The area of X is A if the height of X is H and the width of X is W and A is the result of multiplying H and W.

```
goal(X,A) :- height(X,H) & width(X,W) & evaluate(times(H,W),A)
```

In logic programming languages that provide such predefined concepts, there are usually syntactic restrictions on their use. For example, if a query contains a subgoal with a comparison relation (such as same and distinct), then every variable that occurs in that subgoal must occur in at least one positive literal in the body and that occurrence must precede the subgoal with the comparison relation. If a query uses evaluate in a subgoal, then any variable that occurs in the first argument of that subgoal must occur in at least one positive literal in the body and that occurrence must precede the subgoal with the arithmetic relation. Details are typically found in the documentation of systems that supply such built-in concepts.

In practical logic programming languages, it is also common to include predefined aggregate operators, such as setofall and countofall.

Aggregate operators are typically represented as relations with special syntax. For example the following rule uses the countofall operator to request the number of a person's children. N is the number of children of X if and only if N is the count of all Y such that X is the parent of Y.

```
goal(X,N) :- person(X) & evaluate(countofall(Y,parent(X,Y)),N)
```

As with special relations, there are syntactic restrictions on their use. In particular, aggregate subgoals must be safe in that all variables in the second argument must be included in the first argument or must be used within positive subgoals of the rule containing the aggregate.

3.6 EXAMPLE – KINSHIP

Consider a variation of the Kinship application introduced in Chapter 2. In this case, our vocabulary consists of symbols (representing people) and a binary predicate `parent` (which is true of two people if and only if the person specified as the first argument is the parent of the person specified as the second argument).

Given data about parenthood expressed using this vocabulary, we can write queries to extract information about other relationships as well. For example, we can find grandparents and grandchildren by writing the query shown below. A person `X` is the grandparent of a person `Z` if `X` is the parent of a person `Y` and `Y` is the parent of `Z`. The variable `Y` here is a *thread variable* that connects the first subgoal to the second but does not itself appear in the head of the rule.

```
goal(X,Z) :- parent(X,Y) & parent(Y,Z)
```

In general, we can write queries with multiple rules. For example, we can collect all of the people mentioned in our dataset by writing the following multi-rule query. In this case the conditions are disjunctive (at least one must be true), whereas the conditions in the grandfather case are conjunctive (both must be true).

```
goal(X) :- parent(X,Y)
goal(Y) :- parent(X,Y)
```

In some cases, it is helpful to use built-in relations in our queries. For example, we can ask for all pairs of people who are siblings by writing the query rule shown below. We use the `distinct` condition here to avoid listing a person as his own sibling.

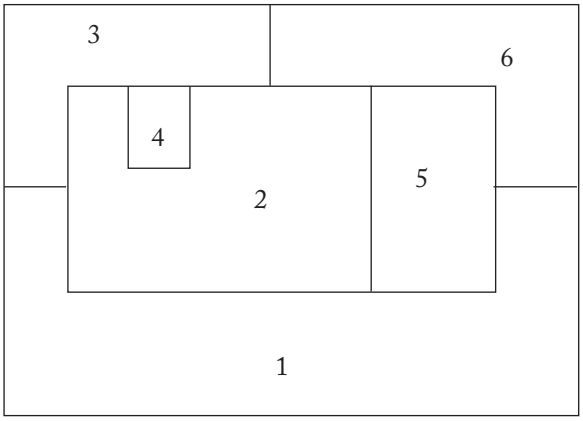
```
goal(Y,Z) :- parent(X,Y) & parent(X,Z) & distinct(Y,Z)
```

While we can express many common kinship relationships using our query language, there are some relationships that are just too difficult. For example, there is no way to ask for all ancestors of a person (parents, grandparents, great grandparents, and so forth). For this, we need the ability to write *recursive* queries. We show how to write such queries in the chapter on *view* definitions.

3.7 EXAMPLE – MAP COLORING

Consider the problem of coloring planar maps using only four colors, the idea being to assign each region a color so that no two adjacent regions are assigned the same color.

A typical map is shown below. Here we have six regions. Some are adjacent to each other, meaning that they cannot be assigned the same color. Others are not adjacent, meaning that they can be assigned the same color.



We can enumerate the hues to be used as shown below. The constants `red`, `green`, `blue`, and `purple` stand for the hues red, green, blue, and purple, respectively.

```
hue(red)
hue(green)
hue(blue)
hue(purple)
```

In the case of the map shown above, our goal is to find six hues (one for each region of the map) such that no two adjacent regions have the same hue. We can express this goal by writing the query shown below.

```
goal(C1,C2,C3,C4,C5,C6) :-
  hue(C1) & hue(C2) & hue(C3) & hue(C4) & hue(C5) & hue(C6) &
  distinct(C1,C2) & distinct(C1,C3) & distinct(C1,C5) & distinct(C1,C6) &
  distinct(C2,C3) & distinct(C2,C4) & distinct(C2,C5) & distinct(C2,C6) &
  distinct(C3,C4) & distinct(C3,C6) & distinct(C5,C6)
```

Evaluating this query will result in 6-tuples of hues that ensure that no two adjacent regions have the same color. In problems like this one, we usually want only one solution rather than all solutions. However, finding even one solution is such cases can be costly. In Chapter 4, we discuss ways of writing such queries that makes the process of finding such solutions more efficient.

3.8 EXERCISES

3.1. For each of the following strings, say whether it is a syntactically legal query.

(a) `goal(X) :- p(a,f(f(X)))`

(b) `goal(X,Y) :- p(X,Y) & ~p(Y,X)`

(c) `~goal(X,Y) :- p(X,Y) & p(Y,X)`

(d) `goal(P,Y) :- P(a,Y)`

(e) `goal(X) :- p(X,b) & p(X,p(b,c))`

3.2. Say whether each of the following queries is safe.

(a) `goal(X,Y) :- p(X,Y) & p(Y,X)`

(b) `goal(X,Y) :- p(X,Y) & p(Y,Z)`

(c) `goal(X,Y) :- p(X,X) & p(X,Z)`

(d) `goal(X,Y) :- p(X,Y) & ~p(Y,Z)`

(e) `goal(X,Y) :- p(X,Y) & ~p(Y,Y)`

3.3. What is the result of evaluating the query `goal(X,Z) :- p(X,Y) & p(Y,Z)` on the dataset shown below.

```
p(a,b)
p(a,c)
p(b,d)
p(c,d)
```

3.4. Assume we have a dataset with a binary predicate `parent` (which is true of two people if and only if the person specified as the first argument is the parent of the person specified as the second argument). Write a query that defines the property of being childless. Hint: use the aggregate operator `countofall`. And be sure your query is safe. (This exercise is not difficult, but it is slightly tricky.)

3.5. For each of the following problems, write a query to solve the problem. Values should include just the digits 8, 1, 4, 7, 3 and each digit should be used at most once in the solution of each puzzle. Your query should express the problem as stated, i.e., you should not first solve the problem yourself and then have the query simply return the answer.

(a) The product of a 1-digit number and a 2-digit number is 284.

(b) The product of two 2-digit numbers plus a 1-digit number is 3,355.

(c) The product of a 3-digit number and a 1-digit number minus a 1 digit number is 1,137.