Artificial Intelligence through Prolog by Neil C. Rowe

Prentice-Hall, 1988, ISBN 0-13-048679-5

Full text of book (without figures)

Table of contents

10. Implementing search

11. Abstraction in search

Preface

Artificial intelligence is a hard subject to learn. I have written a book to make it easier. I explain difficult concepts in a simple, concrete way. I have organized the material in a new and (I feel) clearer way, a way in which the chapters are in a logical sequence and not just unrelated topics. I believe that with this book, readers can learn the key concepts of artificial intelligence faster and better than with other books. This book is intended for all first courses in artificial intelligence at the undergraduate or graduate level, requiring background of only a few computer science courses. It can also be used on one's own.

Students often complain that while they understand the terminology of artificial intelligence, they don't have a gut feeling for what's going on or how you apply the concepts to a situation. One cause is the complexity of artificial intelligence. Another is the unnecessary baggage, like overly formal logical calculi, that some books and teachers saddle students with. But an equally important cause is the often poor connection made between abstract concepts and their use. So I considered it essential to integrate practical programming examples into this book, in the style of programming language and data structures books. (I stress *practical*, not missionaries and cannibals, definitions of "grandfather", or rules for identifying animals in zoos--at least rarely.) This book has about 500 chunks of code. Clear, concrete formalization of artificial intelligence ideas by programs and program fragments is all the more critical today with commercialization and media discovery of the field, which has caused a good deal of throwing around of artificial intelligence terms by people who don't understand them.

But artificial intelligence is a tool for complex problems, and its program examples can easily be forbiddingly complicated. Books attempting to explain artificial intelligence with examples from the programming language Lisp have repeatedly demonstrated this. But I have come to see that the fault lies more with Lisp than with artificial intelligence. Lisp has been the primary language of artificial intelligence for many years, but it is a low-level language, too low for most students. Designed in the early 1960s, Lisp reflects the then-primitive understanding of good programming, and requires the programmer to worry considerably about actual memory references (pointers). Furthermore, Lisp has a weird, hard-to-read syntax unlike that of any other programming language. To make matters worse, the widespread adoption of Common Lisp as a de facto standard has discouraged research on improved Lisps.

Fortunately there is an alternative: Prolog. Developed in Europe in the 1970s, the language Prolog has steadily gained enthusiastic converts, bolstered by its surprise choice as the initial language of the Japanese Fifth Generation Computer project. Prolog has three positive features that give it key advantages over Lisp. First, Prolog syntax and semantics are much closer to formal logic, the most common way of representing facts and reasoning methods used in the artificial intelligence literature. Second, Prolog provides automatic backtracking, a feature making for considerably easier "search", the most central of all artificial intelligence techniques. Third, Prolog supports multidirectional (or multiuse) reasoning, in which arguments to a procedure can freely be designated inputs and outputs in different ways in different procedure calls, so that the same procedure definition can be used for many different kinds of reasoning. Besides this, new implementation techniques have given current versions of Prolog close in speed to Lisp implementations, so efficiency is no longer a reason to prefer Lisp.

But Prolog also, I believe, makes teaching artificial intelligence easier. This book is a demonstration. This book is an organic whole, not a random collection of chapters on random topics. My chapters form a steady, logical progression, from knowledge representation to inferences on the representation, to rule-based systems codifying classes of inferences, to search as an abstraction of rule-based systems, to extensions of the methodology, and finally to evaluation of systems. Topics hard to understand like search, the cut predicate, relaxation, and resolution are introduced late and only with careful preparation. In each chapter, details of Prolog are integrated with major concepts of artificial intelligence. For instance, Chapter 2 discusses the kinds of facts about the world that one can put into computers as well as the syntax of Prolog's way; Chapter 3 discusses automatic backtracking as well as Prolog querying; Chapter 4 discusses inference and inheritance as well as the definition of procedures in Prolog; Chapter 5 discusses multidirectional reasoning as well as the syntax of Prolog arithmetic; and so on. This constant tying of theory to practice makes artificial intelligence a lot more concrete. Learning is better motivated since one doesn't need to master a lot of mumbo-jumbo to get to the good stuff. I can't take much of the credit myself: the very nature of Prolog, and particularly the advantages of the last paragraph, make it easy.

Despite my integrated approach to the material, I think I have covered nearly all the topics in ACM and IEEE guidelines for a first course in artificial intelligence. Basic concepts mentioned in those guidelines appear towards the beginning of chapters, and applications mentioned in the guidelines appear towards the ends. Beyond the guidelines however, I have had to make tough decisions about what to leave out--a coherent book is better than an incoherent book that covers everything. Since this is a first course, I concentrate on the hard core of artificial intelligence. So I don't discuss much how humans think (that's psychology), or how human language works (that's linguistics), or how sensor interpretation and low-level visual processing are done (that's pattern recognition), or whether computers will ever really think (that's philosophy). I have also cut corners on hard non-central topics like computer learning and the full formal development of predicate calculus. On the other hand, I emphasize more than other books do the central computer science concepts of procedure calls, variable binding, list processing, tree traversal, analysis of processing efficiency, compilation, caching, and recursion. This is a computer science textbook.

A disadvantage of my integrated approach is that chapters can't so easily be skipped. To partially compensate, I mark some sections within chapters (usually sections towards the end) with asterisks to indicate that they are optional to the main flow of the book. In addition, all of Chapters 7, 10, and 14 can be omitted, and perhaps Chapters 12 and 13 too. (Chapters 7, 10, 13, and 14 provide a good basis for a second course in artificial intelligence, and I have used them that way myself.) Besides this, I cater to the different needs of different readers in the exercises. Exercises are essential to learning the material in a textbook. Unfortunately, there is little consensus about what kind of exercises to give for courses in artificial intelligence. So I have provided a wide variety: short-answer questions for checking basic understanding of material, programming exercises for people who like to program, "play computer" exercises that have the reader simulate techniques described, application questions that have the reader apply methods to new areas (my favorite kind of exercise because it tests real understanding of the material), essay questions, fallacies to analyze, complexity analysis questions, and a few extended projects suitable for teams of students. There are also some miscellaneous questions drawing on the entire book, at the end of Chapter 15. Answers to about one third of the exercises are provided in Appendix G, to offer readers immediate feedback on their understanding, something especially important to those tackling this book on their own.

To make learning the difficult material of this book even easier, I provide other learning aids. I apportion the book into short labeled sections, to make it easier for readers to chunk the material into mind-sized bites. I provide reinforcement of key concepts with some novel graphical and tabular displays. I provide "glass box" computer programs (that is, the opposite of "black box") for readers to study. I mark key terms in boldface where they are defined in the text, and then group these terms into keyword lists at the end of every chapter. I give appendices summarizing the important background material needed for this book, concepts in logic, recursion, and data structures. In other appendices, I summarize the Prolog dialect of the book, make a few comments on Micro-Prolog, and provide a short bibliography (most of the artificial intelligence literature is now either too hard or too easy for readers of this book). The major programs of the book are available on tape from the publisher for a small fee. Also, I have prepared an instructor's manual.

It's not necessary to have a Prolog interpreter or compiler available to use this book, but it does make learning easier. This book uses a limited subset of the most common dialect of Prolog, the "standard Prolog" of *Programming in Prolog* by Clocksin and Mellish (second edition, Springer-Verlag, 1984). But most exercises do not require programming.

I've tried to doublecheck all examples, programs, and exercises, but some errors may have escaped me. If you find any, please write me in care of the publisher, or send computer mail to rowe@nps-cs.arpa.

Acknowledgements

Many people contributed ideas to this book. Michael Genesereth first suggested to me the teaching of introductory artificial intelligence in a way based on logic. David H. Warren gradually eroded my skepticism about Prolog. Harold Abelson and Seymour Papert have steered my teaching style towards student activity rather than passivity.

Judy Quesenberry spent many long hours helping me with the typing and correction of this book, and deserves a great deal of thanks, even if she ate an awful lot of my cookies. Robert Richbourg has been helpful in many different ways, in suggesting corrections and improvements and in testing out some of the programs, despite his having to jump through all the hoops Ph.D. students must jump through. Richard Hamming provided valuable advice on book production. Other people who provided valuable

comments include Chris Carlson, Daniel Chester, Ernest Davis, Eileen Entin, Robert Grant, Mike Goyden, Kirk Jennings, Grace Mason, Bruce MacLennan, Norman McNeal, Bob McGhee, James Milojkovic, Jim Peak, Olen Porter, Brian Rodeck, Derek Sleeman, Amnon Shefi, and Steve Weingart. Mycke Moore made the creative suggestion that I put a lot of sex into this book to boost sales.

Besides those named, I am grateful to all my students over the years at the Massachusetts Institute of Technology, Stanford University, and the Naval Postgraduate School for providing valuable feedback. They deserve a good deal of credit for the quality of this book--but sorry, people, I'm poor and unwilling to share royalties.

To the reader

Artificial intelligence draws on many different areas of computer science. It is hard to recommend prerequisites because what you need to know is bits and pieces scattered over many different courses. At least two quarters or semesters of computer programming in a higher-level language like Pascal is strongly recommended, since we will introduce here a programming language several degrees more difficult, Prolog. If you can get programming experience in Prolog, Lisp, or Logo that's even better. It also helps to have a course in formal logic, though we won't use much of the fancy stuff they usually cover in those courses; see Appendix A for what you do need to know. Artificial intelligence uses sophisticated data structures, so a data structures course helps; see Appendix C for a summary. Finally, you should be familiar with recursion, because Prolog is well suited to this way of writing programs. Recursion is a difficult concept to understand at first, but once you get used to it you will find it easy and natural; Appendix B provides some hints.

Solving problems is the best way to learn artificial intelligence. So there are lots of exercises in this book, at the ends of chapters. Please take these exercises seriously; many of them are hard, but you can really learn from them, much more than by just passively reading the text. Artificial intelligence is difficult to learn, and feedback really helps, especially if you're working on your own. (But don't plan to do all the exercises: there are too many.) Exercises have code letters to indicate their special features:

> --the code R means a particularly good problem recommended for all readers;
> --the code A means a question that has an answer in Appendix G;
> --the code H means a particularly hard problem;
> --the code P means a problem requiring actual programming in Prolog;
> --the code E means an essay question;
> --the code G means a good group project.

In addition to exercises, each chapter has a list of key terms you should know. Think of this list, at the end of the text for each chapter, as a set of "review questions".

The symbol "*" on a section of a chapter means optional reading. These sections are either significantly harder than the rest of the text, or significantly far from the core material.

Go to book index

Introduction

What artificial intelligence is about

Artificial intelligence is the getting of computers to do things that seem to be intelligent. The hope is that more intelligent computers can be more helpful to us--better able to respond to our needs and wants, and more clever about satisfying them.

But "intelligence" is a vague word. So artificial intelligence is not a well-defined field. One thing it often means is advanced software engineering, sophisticated software techniques for hard problems that can't be solved in any easy way. Another thing it often means is nonnumeric ways of solving problems, since people can't handle numbers well. Nonnumeric ways are often "common sense" ways, not necessarily the best ones. So artificial-intelligence programs--like people--are usually not perfect, and even make mistakes.

Artificial intelligence includes:

> --Getting computers to communicate with us in human languages like English, either by printing on a computer terminal, understanding things we type on a computer terminal, generating speech, or understanding our speech (*natural language*);
>
> --Getting computers to remember complicated interrelated facts, and draw conclusions from them (*inference*);
>
> --Getting computers to plan sequences of actions to accomplish goals (*planning*);
>
> --Getting computers to offer us advice based on complicated rules for various situations (*expert systems*);
>
> --Getting computers to look through cameras and see what's there (*vision*);
>
> --Getting computers to move themselves and objects around in the real world (*robotics*).

We'll emphasize inference, planning, and expert systems in this book because they're the "hard core" of artificial intelligence; the other three subareas are getting quite specialized, though we'll mention them too from time to time. All six subareas are hard; significant progress in any will require years of research. But we've already had enough progress to get some useful programs. These programs have created much interest, and have stimulated recent growth of the field.

Success is hard to measure, though. Perhaps the key issue in artificial intelligence is *reductionism*, the degree to which a program fails to reflect the full complexity of human beings. Reductionism includes how often program behavior duplicates human behavior and how much it differs when it does differ. Reductionism is partly a moral issue because it requires moral judgments. Reductionism is also a social issue because it relates to automation.

Understanding artificial intelligence

Artificial intelligence techniques and ideas seem to be harder to understand than most things in computer science, and we give you fair warning. For one thing, there are lots of details to worry about. Artificial intelligence shows best on complex problems for which general principles don't help much, though there are a few useful general principles that we'll explain in this book. This means many examples in this book are several pages long, unlike most of the examples in mathematics textbooks.

Complexity limits how much the programmer can understand about what is going on in an artificial-intelligence program. Often the programs are like simulations: the programmer sets conditions on the behavior of the program, but doesn't know what will happen once it starts. This means a different style of programming than with traditional higher-level languages like Fortran, Pascal, PL/1, and Ada | REFERENCE 1|, .FS | REFERENCE 1| A trademark of the U.S. Department of Defense, Ada Joint Program Office. .FE where successive refinement of a specification can mean we know what the program is doing at every level of detail. But artificial-intelligence techniques, even when all their details are hard to follow, are often the only way to solve a difficult problem.

Artificial intelligence is also difficult to understand by its content, a funny mixture of the rigorous and the unrigorous. Certain topics are just questions of style (like much of Chapters 2, 6, and 12), while other topics have definite rights and wrongs (like much of Chapters 3, 5, and 11). Artificial intelligence researchers frequently argue about style, but publish more papers about the other topics. And when rigor is present, it's often different from that in the other sciences and engineering: it's not numeric but *logical*, in terms of truth and implication.

Clarke's Law says that all unexplained advanced technology is like magic. So artificial intelligence may lose its magic as you come to understand it. Don't be discouraged. Remember, genius is 5% inspiration and 95% perspiration according to the best figures, though estimates vary.

Preview

This book is organized around the important central ideas of artificial intelligence rather than around application areas. We start out (Chapters 2-5) by explaining ways of storing and using knowledge inside computers: facts (Chapter 2), queries (Chapter 3), rules (Chapter 4), and numbers and lists (Chapter 5). We examine rule-based systems in Chapters 6-8, an extremely important subclass of artificial intelligence programs. We examine search techniques in Chapters 9-11, another important subclass. We address other important topics in Chapters 12-14: Chapter 12 on frame representations extends Chapter 2, Chapter 13 on long queries extends Chapter 3, and Chapter 14 on general logical reasoning extends Chapter 4. We conclude in Chapter 15 with a look at evaluation and debugging of artificial intelligence programs; that chapter is recommended for everyone, even those who haven't read all the other chapters. To help you, appendices A-C review material on logic, recursion, and data structures respectively. Appendix D summarizes the Prolog language subset we use in this book, Appendix E summarizes the Micro-Prolog dialect, Appendix F gives a short bibliography, and Appendix G provides answers to some of the exercises.

Keywords:

artificial intelligence
natural language
inference
planning
expert systems
vision
robotics
reductionism

Go to book index

Representing facts

If we want computers to act intelligent, we must help them. We must tell them all the common-sense *knowledge* we have that they don't. This can be hard because this knowledge can be so obvious to us that we don't realize that a computer doesn't know it too, but we must try.

Now there are many different kinds of knowledge. Without getting deep into philosophy (or specifically *epistemology*, the theory of knowledge), there are two main kinds: facts and reasoning procedures. Facts are things true about the world, and reasoning procedures (or *inferences*) are ways to follow reasoning chains between facts. Since facts are easier to represent than procedures, we'll consider them first, and postpone procedures to Chapter 4.

Predicates and predicate expressions

To talk about facts we need a "language". Artificial intelligence uses many languages and sub-languages. But in this introductory book we don't want to confuse you. We'll use only one, simple (*first-order*) *predicate logic* (sometimes called *predicate calculus* and sometimes just *logic*). And we'll use a particular notation compatible with the computer programming language Prolog | REFERENCE 1|. .FS | REFERENCE 1| In this book we use a subset of the "standard Prolog" in Clocksin and Mellish, *Programming in Prolog*, second edition, Springer-Verlag, 1984. For a complete description of what we use, see Appendix D. .FE Prolog isn't predicate logic itself; computer languages try to do things, whereas logic just says that certain things are true and false. But Prolog does appear close to the way logic is usually written. That is, its *grammar* or *syntax* or form is that of logic, but its *semantics* or meaning is different.

And what is that grammar? Formally, a *predicate expression* (or *atomic formula*, but that sounds like a nuclear weapons secret) is a name--a *predicate*--followed by zero or more arguments enclosed in parentheses and separated by commas (see Figure 2-1) | REFERENCE 2|. .FS | REFERENCE 2| Several terms closely related to "predicate expression" are used in the logic and artificial-intelligence literature. A *literal* is like a predicate expression only it can have a negation symbol in front of it (negations will be explained in Section 3.6). A *structure* or *compound term* is like a predicate expression only it isn't necessarily only true or false. A *logical formula* is a structure or a set of structures put together with "and"s, "or"s, and "not"s. .FE Predicate names and arguments can be composed of any mixture of letters and numbers, except that predicate names must start with a lower-case letter. (Upper-case letters first in a word have a special meaning in Prolog, as we'll explain shortly.) The underscore symbol "_" also counts as a letter, and we will often use it to make names more readable. So these are all predicate expressions:

```
p(x)
q(y,3)
```

```
r(alpha,-2584,beta)
city(monterey,california)
tuvwxy(abc345)
noarguments
pi(3.1416)
long_predicate_name(long_argument_name,3)
```

We can put predicate expressions like these into computers. They can represent facts true about the world. But what exactly do these expressions *mean* (their *semantics*)? Actually, anything you want--it's up to you to assign reasonable and consistent interpretations to the symbols and the way they're put together, though there are some conventions. The better job you do, the more reasonable the conclusions you'll reach from all these facts.

Predicates indicating types

Predicates can mean many things. But they do fall into categories. We summarize the major categories in Figure 2-2.

One thing they can mean is something like data-type information in a language like Pascal or Ada. Except that in artificial intelligence there are generally a lot more types than there are in most programming, because there must be a type for every category in the world that we want the computer to know about.

For instance, suppose we want the computer to know about some U.S. Navy ships | REFERENCE 3|. We .FS | REFERENCE 3| The occasional use of military examples in this book is deliberate: to serve as a reminder that much artificial intelligence work in the United States has been, and remains, supported by the military. We make no endorsements. .FE could tell it

```
ship(enterprise).
```

to say that the Enterprise is a ship (remember we must use lower case). Or in other words, the Enterprise is an example of the "ship" type. We will put periods at the end of facts because Prolog uses the period to signal the end of a line. We could also tell the computer

```
ship(kennedy).
ship(vinson).
```

to give it the names of two more ships--two more things of the "ship" type. Here **ship** is a *type predicate*. If we knew code numbers for planes we could tell the computer about them too, using the code numbers as names:

```
plane(p54862).
plane(p79313).
```

Similarly, we can label people with types:

```
commodore(r_h_shumaker).
president(r_reagan).
```

and label more abstract things like institutions:

```
university(naval_postgraduate_school).
university(stanford_university).
```

and label concepts:

```
day_of_week(monday).
day_of_week(tuesday).
day_of_week(wednesday).
```

A thing can have more than one type. For instance:

```
ship(enterprise).
american(enterprise).
```

And types can have subtypes:

```
carrier(vinson).
ship(carrier).
```

These are all *type predicates*, and they are all have one argument. The argument is the name of some thing in the world, and the predicate name is the class or category it belongs to. So the predicate name is *more general* than the argument name; this is usual for predicate names in artificial intelligence. So it wouldn't be as good to say

```
enterprise(ship).
kennedy(ship).
```

About types

We've said these predicates are like the types in computer languages, but there are some differences. The main one is that they need never be defined anywhere. If for instance we are using Pascal, we either use the built-in types (integer, real, character, array, and pointer) or define the type we want in terms of those built-in types. But for artificial intelligence, the type (predicate) names are just arbitrary codes used in lookup. This is because you can put integers and characters in a computer, but not a ship. You can't even put in a full representation of a ship, or a full representation of any other real object--real objects have too many complexities, while integers and characters are abstractions.

How then, if we expect the computer to be intelligent, will it ever know what a ship is? Much the way people know. Ships are defined in a dictionary using the concept of a vehicle, the concept of water, the concept of floating, and so on. A dictionary might say a ship is "an oceangoing vessel". But it might define "vessel" as a "craft for travelling on water", and "craft" as an "individual ship"--so the definitions are circular, as all dictionary definitions are sooner or later. But we can indirectly figure out what is being talked about by the secondary words like "oceangoing" and "travelling". So words must be defined in terms of one another.

So we won't expect each type predicate to be *implemented* (that is, understood by a computer) by a separate procedure or processing routine. The same holds for arguments. In fact, we could store all predicate names and arguments the same way in the computer, as characters. This is a bit wasteful of computer storage space--so some Prolog dialects do store numbers differently--but there's nothing wrong philosophically with it.

Good naming

So predicate and argument names can be arbitrary; we just have to remember what they represent. But one name can be better than another, if it is easier to remember what it means. Writing facts for an artificial-intelligence program to use is a kind of programming, and we should follow the usual rules of good programming style. In choosing names, we suggest these guidelines:

1. As much as possible, use everyday English words for names. If you need more than one word, use the underscore character between them for clarity, like in **day_of_week** (though sometimes you can leave out the underscores like in **dayofweek** when the reading is reasonably clear).

2. Choose names that describe their function precisely. For instance, use **day_of_week** instead of **day**, which could describe both **monday** and **october_19_1985**.

3. Avoid names with multiple meanings. For instance, if there is a Commander Kennedy as well as a ship named Kennedy, include the first initial of the person; or if you call the Enterprise a "ship", don't also say that a unit "shipped" somewhere.

4. Avoid numbers in names, with two exceptions: arithmetic (see Chapter 5) and closely related variables and predicates (like **X** and **X2** in Section 5.5 and **iterate** and **iterate2** in Section 10.8).

5. Abbreviate only when absolutely necessary. Since artificial intelligence programs often use many names, abbreviations can be confusing.

6. Predicate names should be more general than their argument names, but not so general that they don't really mean anything (for then facts can't be indexed well).

7. A few predicate names are reserved or "special" to Prolog, so you can't use them for your own predicates.

8. Of course, always use the same name for the same thing.

Property predicates

We can tell the computer (or *assert*):

```
ship(enterprise).
gray(enterprise).
big(enterprise).
```

which could mean "The Enterprise is a ship, it is gray, and it is big." (Never mind that "big" is vague; we could define it as "more than 800 feet long", and "gray" and even "ship" are vague to a lesser extent (is a toy ship a ship? and is an imaginary ship a ship?), and much human knowledge is vague anyway.) Or this could mean the Enterprise is a member of the class of ships, a member of the class of gray things, and a member of the class of big things. But those last two phrases awkward. "Gray" and "big" are adjectives, not nouns like "ship", and they should be treated differently.

So we'll represent properties of objects as *property predicate expressions*, two-argument expressions in which the predicate name is the name of a property, the first argument is the name of an object, and the second argument is the value of the property. The preceding example could be rewritten better as:

```
ship(enterprise).
color(enterprise,gray).
size(enterprise,big).
```

This has the advantage of using predicate names that are more general. It also shows the relation between **gray** and **enterprise**, and that between **big** and **enterprise**: **color** and **size** are the property names for which **gray** and **big** are the values. So we've made some implicit (unstated "common-sense") knowledge explicit (stated), a key goal in artificial intelligence.

Again, the computer won't actually know what **gray** and **big** mean if we type in the preceding three example lines; those are just codes that it uses for comparison. For instance, if the computer also knows

```
color(kennedy,gray).
```

then it knows the Enterprise and the Kennedy have the same color, though it doesn't know what a "color" is (but don't blame it, because most computers don't have eyes).

An important class of property predicates concern space and time. For instance

```
location(enterprise,14n35e).
last_docking(enterprise,16feb85).
```

could mean that the Enterprise is currently at latitude 14N and longitude 35E, and its last docking was on February 16, 1985.

Predicates for relationships

Perhaps the most important predicates of all relate two different things. Such *relationship predicates* are important because a lot of human reasoning seems to use them--people need to relate ideas. For instance, we can use a **part_of** predicate of two arguments which says that its first argument is a component within its second argument. We could give as facts:

```
part_of(enterprise,u_s_navy).
part_of(u_s_navy,u_s_government).
part_of(naval_postgraduate_school,u_s_government).
part_of(propulsion_system,ship).
```

In other words, the Enterprise is part of the U.S. Navy, the Navy is part of the U.S. government, the Naval Postgraduate School is part of the U.S. government, and the propulsion system is part of a ship. An **owns** relationship predicate can say that something is owned by someone:

```
owns(tom,fido).
owns(tom,toms_car).
```

These facts say that Tom owns two things: something called **fido**, and an unnamed car which we can just refer to as **toms_car**.

It's easy to get confused about argument order in relationship predicate expressions. So we'll try to follow this convention: if the predicate name is inserted between the two arguments, the result will be close to an English sentence giving the correct meaning. So if we insert "owns" between "tom" and "fido" we get "Tom owns Fido", and if we insert "part of" between "enterprise" and "u. s. navy" we get "Enterprise part of U. S. Navy".

An important class of relationship predicates relates things in space and time. A real-world object can be north, south, east, west, etc. of another object. Viewed by a fixed observer, an object can also be right, left, above, below, in front, and behind another object. We can describe a picture with these predicates. Similarly, an event in time can be before, after, during, overlapping, or simultaneous with another event, so we can describe history with these predicates.

Relationship predicates can describe relationships between people. For instance, the **boss_of** relationship is important for describing bureaucracies, an important application of artificial intelligence. It says that a person (first argument) is the boss of another person (second argument), and this shows direction of responsibility. People can also be related by kinship relationship predicates (**father**, **mother**, **child**, **uncle**, **cousin**, **stepfather**, **half-brother**, **grandfather**, etc.). People can also be related with **friend** and **acquaintance** relationship predicates.

Besides all these, another special relationship predicate is frequently used in artificial intelligence. It's called **a_kind_of** or **is_a** (we prefer the first name, because "is" is vague), and it can replace all type predicates. Its first argument is a thing, and its second argument is the type of that thing (the predicate name in the one-argument form considered before). For instance:

```
a_kind_of(enterprise,ship).
a_kind_of(tanker,ship).
a_kind_of(tuesday,day_of_week).
```

which says that the Enterprise is a kind of ship, a tanker is a kind of ship, and Tuesday is a kind of day of the week | REFERENCE 4|. .FS | REFERENCE 4| Some researchers don't agree with this use of **a_kind_of**. They think that the first two facts should have different predicate names since the Enterprise is an individual while tankers are a group of individuals; often they'll use the predicate name **element** for the "Enterprise" fact, and keep **a_kind_of** for the "tanker" fact. But a set whose size is 1 is still a set, and there doesn't seem to be anything fundamentally different between restricting the body type of a ship to be a tanker and restricting the name of a ship to be the word "Enterprise"--it just happens that people try, not always successfully, to make names unique. Researchers who argue against this may be getting this issue confused with the important "extensions vs. intensions" problem which we'll discuss in Section 12.8. .FE Some reasoning is easier with this two-argument form than the equivalent one-argument form.

There are other predicates, but as any psychotherapist will tell you, relationships are the key to a happy life.

Semantic networks

Pictures can make a complicated set of facts a lot clearer. There's a simple pictorial way to show the predicate expressions we've been discussing: the *semantic network*. Unfortunately, there is a major restriction: semantic networks can only directly represent predicates of two arguments (so type predicates must be in the two-argument form) | REFERENCE 5|. .FS | REFERENCE 5| But we can represent predicate expressions with more than two arguments indirectly, as sets of two-argument predicate expressions. .FE

A semantic network is what computer scientists call a a *labeled directed graph* (see Appendix C for a definition). We make every possible fact argument a small named circle (node) in the graph. For each two-argument fact, we draw an arrow (edge) from the circle for its first argument to the circle for its second argument, and label the arrow with the predicate name. So the fact **p(a,b)** is represented as an arrow from a circle labeled "a" to a circle labeled "b", with the arrow itself labeled "p". If for instance our facts are:

```
a_kind_of(enterprise,ship).
a_kind_of(kennedy,ship).
part_of(enterprise,u_s_navy).
part_of(kennedy,u_s_navy).
part_of(u_s_navy,u_s_government).
a_kind_of(u_s_government,government).
color(ship,gray).
location(enterprise,15n35e).
has(u_s_government,civil_service_system).
```

then our semantic network looks like Figure 2-3.

Getting facts from English descriptions

Usually programmers building artificial intelligence programs don't make up facts themselves. Instead, they look up facts in documents and books, or ask people knowledgeable about the subject ("experts") to tell them what they need--the process of *knowledge acquisition*. But English and other "natural languages" are less precise than computer languages (though more flexible), and the programmer must be careful to get the meanings right.

The sorts of facts we've considered so far are usually often signalled by the verb "to be" in English (**is**, "are", "was", "were", "will be", "being", and so on). For instance:

> The Enterprise is a ship.
> A ship is a vehicle.
> The Enterprise is part of the U.S. Navy.
> A ship is gray.

Here "to be" is used for type predicates (the first and second sentences), a **part_of** relationship predicate (the third sentence), and a property predicate (the fourth sentence). Plurals can also be used:

> Ships are gray.

> Ships are vehicles.

English verbs with narrower meanings can be used too:

> The Enterprise belongs to the U.S. Navy.
> The Enterprise has a hull.
> They color ships gray.
> The Enterprise is located at 15N35E.

The first two suggest **part_of** relationship predicates, and the last two are property predicates.

Predicates with three or more arguments

You can have as many arguments to a predicate as you want if you're not concerned about easily representing them in a semantic network. One idea is to include multiple property values in a single fact, much like adjectives and adverbs modifying a noun or verb. So for instance we could put everything we know about a ship together:

```
ship_info(enterprise,15n35e,1200,16feb85,gray,j_kirk).
```

which we could read as "The Enterprise is a ship that was at 15N25E at 12 noon on February 16, 1985, and its color is gray, and its captain is J. Kirk." To interpret such facts we need to keep a description somewhere of the properties and their order within the arguments.

These sort of predicates define a *relational database* of facts. Much research has studied efficient implementation and manipulation of such databases. The information about properties and their order for each such predicate is called a *database schema*.

Another important category of predicates with often many arguments (though they can also have just two) is that representing results of actions--in mathematical terminology, *functions*. Suppose we want to teach a computer about arithmetic. We could use a predicate **sum** of three numerical arguments, which says that the sum of the first two arguments is the third. We could give as facts:

```
sum(1,1,2).
sum(1,3,4).
sum(1,4,5).
sum(1,5,6).
sum(2,1,3).
sum(2,2,4).
sum(2,3,5).
sum(2,4,6).
```

And we could do this for lots of different numbers, and different arithmetic operations. Of course for this to be useful in general, we would need very many facts and this would be unwieldy (we will describe a better way in Chapter 5), but it will suffice to define operations on any finite set of numbers.

We will use function predicates frequently. To avoid confusion, we follow the convention that the last argument always represents the result of (value returned by) the function, with the exception noted in the next section | REFERENCE 6|. .FS | REFERENCE 6| If you're familiar with Lisp, be careful to include the function result as an argument to Prolog predicates. In Lisp, a value is always associated with the whole expression, something you can't do in Prolog. .FE

Functions can also be nonnumeric. An example is a function that gives, for two employees of a company, the name of the lowest-ranking boss over both of them. Since artificial intelligence emphasizes nonnumeric reasoning, you'll see more nonnumeric than numeric functions in this book.

Probabilities

We have assumed so far that facts are always completely certain. In many situations (as when facts are based on reports by people), facts are only probably true. Then we will use the mathematical idea of *probability*, the expected fraction of the time something is true. We will put an approximate probability as an optional last argument to a predicate, after the previously discussed function result if any. So for instance

```
color(enterprise,gray,0.8).
```

says that we're 80 percent sure (or sure with probability 0.8) that the Enterprise is gray. We'll ignore this topic until Chapter 8.

How many facts do we need?

An infinity of facts are true about the world. How then do we decide which to tell a computer? This question has no easy answers. Generally, you must decide what you want the computer to do. Then make sure to tell the computer in advance every fact that might be relevant to that behavior. "Libraries" of useful facts for particular subjects will help. But the smarter you want the computer to be, the more facts you must tell it. The next chapter will discuss the next question, how to get the computer to do things with facts.

Keywords:

knowledge
facts

logic
predicate calculus
Prolog
predicate
predicate expression
arguments
semantics
type predicate
assertion
property predicate
relationship predicate
part_of
a_kind_of
semantic network
knowledge acquisition
relational-database predicate
function predicate
probability predicate

Exercises

(Note: answers to exercises marked with the code "A" are given at the back of the book.)

2-1. (A,E) Which of the following facts is better knowledge representation? Explain. ("Better" means less likely to confuse people.)

```
color(enterprise,gray).
size(enterprise,big).
```

2-2. (R,A,E) Suppose you want to store facts about when and where memos were sent in an organization. Which is the best Prolog format for such facts, and why?

(i) <date>(<name>,<author>,<distribution>).
(ii) memo(<name>,<date>,<author>,<distribution>).
(iii) fact(memo,<name>,<date>,<author>,<distribution>).

2-3. Draw a semantic network representing the following facts:

Ships are things.
Carriers are ships.
Ships have a position.

Ships have a crew.
Carriers have planes.
Planes are things.
A crew consists of people.
People are things.

2-4. Represent the nonnumeric meaning of the picture in Figure 2-4 as a set of nonnumeric Prolog facts. (Hint: describe the circles and their relationships.)

2-5. (A) One-argument and two-argument predicates are very common in Prolog knowledge representation. Most of the facts you want to put into computers can be represented with them. Someone might say that this shouldn't be surprising, because most operations in mathematics are either unary (applied to a single thing like the square root operation or the sine operation), or binary (applied to two things, like addition and exponentiation). What's wrong with this comment?

2-6. (R,A) Consider the six-argument ship_info facts in Section 2.9. To represent them in a semantic network, we need to convert each to a set of two-argument facts. Explain how. Assume that six-argument facts only record the most recent position of a ship.

2-7. (A,E) Why might it be a good idea to put falsehoods (statements false in the world) into a computer?

2-8. Suppose you want to write a program that reasons like Sherlock Holmes did, about the facts of some crime to decide who is responsible. You want to represent in Prolog style the facts you find about the crime, and the reports of witnesses. The argument types used for facts will vary. But certain arguments must be included in every fact about the crime--what are they? And certain arguments must be included in every fact giving a report from a witness--what are they?

2-9. (E) Why must the representation of these two facts be fundamentally different?

Clint is mayor.
Someone is mayor.

2-10. (E) Consider the use of the word "boss" in the following facts. Suppose you wanted to represent these facts in Prolog. Would it be a good idea for any two of these to use the same word "boss" as either a predicate name or an argument name?

Mary is boss of Dick.
Dick and Mary boss their children around.
"Boss" has four letters.
A boss has managerial responsibilities.

2-11. (E) Another way to put facts inside computers is with a restricted subset of English. For instance:

> **The Enterprise is a ship.**
> **The Enterprise is part of the US Navy.**
> **The color of the Enterprise is gray.**
> **The Enterprise is at 15N25A.**

(a) Discuss the advantages of storing facts this way instead of with predicate expressions as we have done in the chapter.

(b) Give a disadvantage for efficient use of the facts.

(c) Give a disadvantage for programming errors.

Go to book index

Variables and queries

We can put facts into a computer. So what can we do with them? Well, we want to *reason* about facts and conclude new facts--what's called *inference*. For this we'll need the concepts of queries, variables, and backtracking.

Querying the facts

One thing we can do with facts in a computer is to look them up. This is the usual mode of *Prolog interpreters*, software that interprets and executes code written in the Prolog language | REFERENCE 1|: they wait for us to give them things they can try to look up. .FS | REFERENCE 1| Most Prolog-understanding software are interpreters like this and not *compilers*. A Prolog interpreter is not an "artificial intelligence program" but a tool to execute artificial-intelligence programs written in the Prolog language. .FE You're in this *query mode* when the Prolog interpreter types **?-** at the front of every line. Query mode is the way database query languages work, like SQL and QUEL.

To make this clearer, assume these facts (the semantic network example from Section 2.7) have been entered into a computer running a Prolog interpreter:

```
a_kind_of(enterprise,ship).
a_kind_of(kennedy,ship).
part_of(enterprise,u_s_navy).
part_of(kennedy,u_s_navy).
part_of(u_s_navy,u_s_government).
a_kind_of(u_s_government,government).
color(ship,gray).
location(enterprise,15n35e).
has(u_s_government,civil_service_system).
```

We call such a set of facts known to a Prolog interpreter a *Prolog database* or just *database*. As we'll explain shortly, databases can be loaded from files. The block diagram in Figure 3-1 summarizes these basics.

Now in query mode we can type

```
part_of(kennedy,u_s_navy).
```

so that what actually shows on the computer terminal will be

```
?- part_of(kennedy,u_s_navy).
```

Note the period; the interpreter will wait (forever if necessary) until we type it. Then the interpreter will type in reply the single word **yes** to acknowledge that the fact is in its database. If we ask instead

```
?- part_of(pequod,u_s_navy).
```

(again the **?-** is typed by the interpreter and not us), the computer will type the single word **no**. So **yes** means "I found it" and **no** means "I couldn't find it". We call a **yes** a query *success* and **no** a query *failure*. So to make the computer say **no** when a query is *false*, the database must include every truth about its subject, for otherwise **no** could mean incomplete data.

Queries with one variable

But this isn't too interesting. A query must give the precise fact we want to look up, including every argument. We might instead want to ask if a **part_of** fact has **enterprise** as its first argument and anything at all as its second argument. We can do this by querying

```
?- part_of(enterprise,X).
```

Read this as "Find me an "X" such that **part_of(enterprise,X)** is true," or simply as "What is the Enterprise part of?" The Prolog interpreter will go through its facts in order, trying to match each to the query. When it finds one that matches in predicate name and first argument, it will type "**X=**" followed by the fact's second argument, instead of typing **yes**. Or in technical jargon, it *binds* or *matches* **X** to a value and prints it. So for this query with the previous database, we will get

```
X=u_s_navy
```

or **X** is bound to **u_s_navy**.

X here is a *variable*. Prolog variables have similarities to variables in other programming languages, but also important differences we'll encounter as we proceed. Prolog variables are designated by a capitalized first letter in a word (followed by other letters and numbers, either capitalized or uncapitalized), and this is why in the last chapter we used lower case for other words in Prolog.

Variables can only be arguments in Prolog; they can't appear as predicate names (though we'll give a way around this limitation in Chapter 12). This means Prolog represents only *first-order logic*. First-order logic is sufficient for nearly all artificial-intelligence applications, so that's no big deal. First-order logic is a reason we insisted in Chapter 2 on predicate names more general than their argument names: variables pay off when they stand for lots of possibilities.

Multi-directional queries

A variable can appear anywhere among the arguments to a predicate expression in a query, with some exceptions to be discussed later. So we could also query with the previous database

```
?- part_of(X,u_s_navy).
```

and the Prolog interpreter will type back

```
X=enterprise
```

In other words, we can be flexible about which arguments are *inputs* (constants) and which are *outputs* (variables) in a query. This means Prolog can answer quite different questions depending on where we put variables in the query. This flexibility extends to calls of Prolog procedures (subroutines and functions) too, as you will see in Chapter 4, a big difference from most programming languages.

Matching alternatives

More than one thing (value) can match (bind) a query variable. The Prolog interpreter will find the first, print it out, and stop and wait If just one is sufficient, type a carriage return. But to see the next answer (if any), type a semicolon ("**;**"). We can keep typing semicolons, and it will keep finding new matches, until it can't find any more and it must answer **no**. So for our example database if we query

```
?- a_kind_of(X,ship).
```

which means "Find me an X that's a kind of ship," the interpreter will first type

```
X=enterprise
```

and then if we type a semicolon it will type

```
X=kennedy
```

and then if we type a semicolon it will type **no**. The semicolon prints at the end of the line, so what this will all look like on the computer terminal will be:

```
?- a_kind_of(X,ship).
X=enterprise;
X=kennedy;
no
```

where we typed the two semicolons and the first line except for the "**?-** ", and the interpreter typed the rest.

We can have more than one variable in a query. If we were to query for our example database

```
?- part_of(X,Y).
```

("What things are part of other things?") and we kept typing semicolons, we would eventually see on the computer terminal some like this (some Prolog dialects format this output slightly differently):

```
X=enterprise, Y=u_s_navy;
X=kennedy, Y=u_s_navy;
X=u_s_navy, Y=u_s_government;
no
```

So semicolons find us every combination of bindings of the variables that satisfies a query. Since the Prolog interpreter works top to bottom through the database, the bindings will reflect database order.

Multi-condition queries

A Prolog interpreter also lets us specify that several different conditions must succeed together in a query. This lets us specify "chains of reasoning", like those so important to detectives in mystery fiction.

Suppose we wanted to know the gray-color of the Enterprise. If we type

```
?- color(enterprise,C).
```

we get **no** with our example database, because the color fact is about ships in general and not the Enterprise. This problem of information in the "wrong place" happens often in artificial-intelligence systems. Instead we can ask if there is some category or type T that the Enterprise belongs to, such that everything of type T has color C:

```
?- a_kind_of(enterprise,T), color(T,C).
```

This represents an "and" (*conjunction*) of two predicate expressions, both of which must succeed for the whole match to succeed. It works this way: we first try to answer the query

```
?- a_kind_of(enterprise,T).
```

Then for that particular **T**, we answer the query

```
?- color(T,C).
```

Using our example database, we first match **T** to **ship** in the first-listed fact. We then look for a **color** fact in which this **T** is the first argument, and the seventh-listed fact qualifies; we can then match **C** to **gray**. The Prolog interpreter now types out:

```
T=ship, C=gray
```

So commas between predicate expressions in a query line mean reusing the same values for any same-named variables. Commas are like a logical "and" since all the subqueries (predicate expressions) must succeed for the whole query to succeed. To make commas easier to spot, we'll often put spaces after them in queries; these spaces are ignored by the interpreter. (But don't put spaces in predicate expressions.)

As another example, suppose we want to know what the Enterprise is part of. We could say

```
?- part_of(enterprise,X).
```

and get **X=u_s_navy**, but that's not the only reasonable answer since the U.S. Navy is part of something else. So we could say:

```
?- part_of(enterprise,X), part_of(X,Y).
```

and get back **X=u_s_navy, Y=u_s_government**.

Logical "or" (*disjunction*) is represented by a semicolon instead of a comma. For instance

```
?- color(enterprise,C); color(ship,C).
```

asks for the color of the Enterprise if any is recorded, otherwise the color of ships in general. Parentheses can group which predicate expressions go with which others in "and"s and "or"s. So for instance the two conditions under which something is part of something else could be compressed into one with:

```
?- part_of(enterprise,X); (part_of(enterprise,Y), part_of(Y,X)).
```

This reads: "Find me an X such that either the Enterprise is part of it, or the Enterprise is part of some Y that is part of it." We won't use these "or" semicolons much, because (1) "and"s occur more often in applications, (2) they often require parentheses and so are hard to read, and (3) there is a better way to get the effect of an "or", to be discussed in the next chapter.

Figure 3-2 should help you keep straight the special symbols we've used so far, plus previewing a few to come.

Negative predicate expressions

So we have "and"s and "or"s. All we need to complete a Boolean algebra is a negation or "not". This is accomplished by the built-in predicate **not** whose one argument is a predicate expression. (A *built-in* predicate is one with special meaning to the interpreter, a meaning not given by facts.) A **not** succeeds whenever querying its argument fails, a fails whenever querying its argument succeeds. So the query

```
?- not(color(enterprise,green)).
```

will succeed whenever there's no fact that the color of the Enterprise is green, and fail when there is such a fact. We'll extend the term "predicate expression" to include such **not** expressions too.

How will the Prolog interpreter ever be sure something is *not* true? Strictly speaking, it can't, since facts that directly say something is false are not permitted in Prolog (Chapter 14 discusses this further). So **not** is defined to mean the interpreter couldn't find a fact in its database--*negation-by-failure* or *the closed-world assumption*. Yet this is a curious and awkward interpretation of "not", not what we usually mean by the word in English. So we must be careful with **not** in Prolog. One big problem is that we can't, with a few exceptions, put unbound variables within a **not**. So this query won't work:

```
?- not(color(X,gray)), a_kind_of(X,ship).
```

(This asks for a ship **X** that isn't gray.) Instead we must reverse the order of the two things:

```
?- a_kind_of(X,ship), not(color(X,gray)).
```

Some query examples

Questions in English about a database often map directly into Prolog queries. Words like **is**, "are", "does", and "did" at the beginning of a question suggest queries without variables (yes/no queries). Words like "what", "which", "who", "where", "when", and "how" suggest variables.

Here are some examples. We assume the meanings of the **part_of**, **color**, **a_kind_of**, etc. predicates we've been assuming all along. (These queries print out additional variable values than those desired; Chapter 4 will explain how to prevent this.)

1. What things are part of gray things?

```
?- part_of(X,Y), color(Y,gray).
```

2. What things are part of parts of other things?

```
?- part_of(A,B), part_of(B,C).
```

3. What things are gray or blue?

```
?- color(T,gray); color(T,blue).
```

4. What isn't an example of a gray thing? (**Example** suggests the reverse of the **a_kind_of** relationship.)

```
?- a_kind_of(E,T), not(color(T,gray)).
```

5. What is the Enterprise, either directly or through one level of indirection?

```
?- a_kind_of(enterprise,J); (a_kind_of(enterprise,K), a_kind_of(K,J)).
```

6. What things of which the Enterprise is part, are themselves part of something that has a civil service system?

```
?- part_of(enterprise,U), part_of(U,V), has(V,civil_service_system).
```

Loading a database

How do we load a database of facts into a Prolog interpreter in the first place? This varies between implementations of Prolog interpreters, but usually we must first enter the facts we want into a text file, using an editor program. We exit the editor and start up the Prolog interpreter. We then query a special *built-in* loading predicate, called **consult** in this book. This **consult** is not something we must give facts for, but an internal Prolog name like **not**; it takes one argument, the name of a file to load into the Prolog interpreter's internal memory (database). From then on, the facts in that file will be used to answer queries.

For instance, suppose we use the editor to create a file called "test", containing:

```
boss(harry).
employee(tom).
employee(dick).
```

We can start the Prolog interpreter, type the query

```
?- consult(test).
```

and then type the query

```
?- employee(X).
```

and get

```
X=tom
```

from the first fact in the file that matches.

We can load more than one file into the Prolog interpreter, if several files contain useful facts. Just query **consult** again. New facts are put after the old facts, so you can get answers in a different order if you load the same files in a different order.

Backtracking

Let's consider in more detail how the Prolog interpreter answers complicated queries. To make this easier, consider for now queries with only commas ("and"s), no semicolons ("or"s) or **not**s.

Predicate expressions "and"ed in a query are first taken left to right. That is, the leftmost expression is tried first, then the second expression from the left (using whatever variable matches were found for the first) and so on. So predicate expressions in a query are initially done in order, like lines of a program in a conventional language like Pascal.

But suppose that a predicate expression fails--that is, no fact matching it can be found. If the expression has variables that were bound earlier in the query line, the fault may just be in the bindings. So the interpreter automatically *backtracks* (goes back to the immediately previous expression in the query) and tries to find a different fact match. If it cannot, then *that* predicate expression fails and the interpreter backtracks to the previous one, and so on.

Anytime the Prolog interpreter cannot find another matching for the leftmost expression in a query, then there's no way the query could be satisfied; it types out the word **no** and stops. Anytime on backtracking it can find a new matching for some predicate expression, it resumes moving right from there as it did originally.

The purpose of backtracking is to give "second chances" to a query, by revising earlier decisions. Backtracking is very important in artificial intelligence, because many artificial-intelligence methods use intelligent guessing and following of hunches. Guesses may be wrong, and backtracking is a good way to recover then.

Here's an example:

```
?- part_of(X,Y), has(Y,civil_service_system).
```

which asks for an **X** that is part of some **Y** that has a civil service system. Assume the standard database example of this chapter. Then the only facts that will help with this query (the only facts with predicate names **part_of** and **has**) are:

```
part_of(enterprise,u_s_navy).
part_of(kennedy,u_s_navy).
part_of(u_s_navy,u_s_government).
has(u_s_government,civil_service_system).
```

Here in detail is what the Prolog interpreter does to answer this query:

> 1. It takes the first predicate expression in the query, and matches **X** to **enterprise**, and **Y** to **u_s_navy**. It stores the information that it has chosen the first fact to match the first expression.
>
> 2. It then moves to the second predicate expression, and tries to answer the *subquery*
>
> ```
> ?- has(u_s_navy,civil_service_system).
> ```
>
> That is, it "substitutes" in the value bound to variable **Y**. But the subquery fails since there's no such fact.
>
> 3. So it must backtrack, or return to the first predicate expression in the query. From its stored information, it knows it chose the first **part_of** fact last time, so now it tries the second, binding **X** to **kennedy** and **Y** to **u_s_navy**. It stores the information about what it chose.
>
> 4. It then tries to answer the subquery
>
> ```
> ?- has(u_s_navy,civil_service_system).
> ```
>
> This is the same query it did in step 2, but the interpreter is stupid and doesn't remember (Chapter 6 will explain how to force it to remember), so it checks the facts and fails to find anything again. The subquery fails.
>
> 5. So it backtracks again, to the first predicate expression in the query. It chose the second fact last time, so it now chooses the third (and last) **part_of** fact. So **X** is bound to **u_s_navy** and **Y** is bound to **u_s_government**.
>
> 6. The second expression is considered with the new binding for **Y**, and the interpreter tries to answer the subquery

```
?- has(u_s_government,civil_service_system).
```

And this succeeds because it's the fourth fact.

7. So both predicate expressions in the query succeed, and the whole query succeeds. The interpreter prints out the bindings that it found:

```
X=u_s_navy, Y=u_s_government
```

Notice that the interpreter wouldn't have had to backtrack if we just reversed the order of the query ("and" is commutative--see Appendix A):

```
?- has(Y,civil_service_system), part_of(X,Y).
```

because only one fact can match the **has** predicate expression. But that requires analyzing the facts in advance, and probably won't be true for a complete database for an application.

The automatic backtracking of the Prolog interpreter has both advantages and disadvantages. A big advantage is that combinatorial problems are easier to specify than with most computer languages, because the interpreter does more work for you. It also means that Prolog is a more flexible language than most: if you refer to an unbound variable in Pascal, Ada, PL/I, or Fortran, you get an error message and the program stops. The disadvantages are that Prolog programs run slower than those of other languages, and they're sometimes harder to understand and debug, because the language tries to do more.

A harder backtracking example: superbosses

Here is another backtracking example. It's trickier than the last because two predicate expressions both have alternatives. Furthermore, the same predicate name is used twice, and we have to distinguish the alternatives for each use.

Suppose we have facts about employees in an organization, represented with a two-argument predicate **boss**. Its first argument a boss, and its second argument is an employee of that boss. Take the following example database:

```
boss(dick,harry).
boss(tom,dick).
boss(ann,mary).
boss(mary,harry).
```

Suppose we want find "superbosses", people who are bosses of bosses. That is, those X that are a boss of some Y while at the same time Y is a boss of some Z. We can issue the query

```
?- boss(X,Y), boss(Y,Z).
```

and every match the interpreter finds for variable **X** will be a superboss. (Matches for **Y** and **Z** will also be found, but **X** is all that we want, according to the way we stated the problem.)

Let's trace query execution (summarized in Figure 3-3). As usual, assume facts are placed in the Prolog database in the order listed.

> 1. The first predicate expression in the query will match the first fact in the database, with **X=dick** and **Y=harry**.
>
> 2. Moving to the second predicate expression in the query, the interpreter searches for a **boss** fact with **harry** as its first argument. But there's no such fact in the database, so the second expression in the query fails.
>
> 3. So the interpreter backtracks, returning to the first expression to make another choice. Last time it used the first fact in the database, so this time it uses the second fact and sets **X=tom** and **Y=dick**.
>
> 4. Things proceed just as if these matchings happened originally. The interpreter goes to the second predicate expression, and searches for a **boss** fact where **dick** is the first argument. And yes, there is such a fact, the first fact in the database.
>
> 5. So **Z=harry**, and since we're at the end of the query, the query succeeds. Therefore Tom is a superboss. The interpreter types out **X=tom, Y=dick, Z=harry**.

Now we can explain better what typing a semicolon does after the Prolog interpreter types out a query answer (not to be confused with a semicolon in a *query:* it forces failure and backtracking. For instance, suppose after that answer **X=tom** we type a semicolon instead of a carriage return. What happens now is summarized in Figure 3-4, together with the previous events.

> 6. The interpreter will go back to what it just finished, the second expression of the query, and try to find a different match.
>
> 7. The old match for the second query expression was from the first fact, so now it examines the second, third, and fourth facts in order. Unfortunately, none have **dick** as their first argument, so the expression fails.
>
> 8. So the interpreter must return to the first predicate expression yet again. The first and second facts have been tried, so it uses the third fact and sets **X=ann** and **Y=mary**.

9. It resumes normal left-to-right processing and tries to find a match for the second query expression, starting at the top of the list of facts. (Each time it approaches a query predicate expression from the left, it starts at the top of the facts.) This means finding a fact where **mary** is the first argument, and indeed the fourth fact qualifies.

10. So **Z=harry**, and the entire query succeeds when **X=ann**, meaning that Ann is a superboss. The interpreter types out **X=ann, Y=mary, Z=harry**.

Backtracking with "not"s

Negated predicate expressions (expressions with a **not**) are easy with backtracking. Since they can't bind variables to succeed, they can be skipped in backtracking. For instance, we could add another expression to our superboss query to insist that the superboss not be the boss of Dick:

```
?- boss(X,Y), not(boss(X,dick)), boss(Y,Z).
```

Then when the interpreter executes the new query (see Figure 3-5):

1. The first predicate expression matches the first fact in the database as before, setting **X=dick** and **Y=harry**.

2. This binding of **X** satisfies the second condition, the **not** (Dick isn't his own boss).

3. For the third expression, there's no fact with **harry** as its first argument, so it fails. The interpreter backtracks to the immediately previous (second) expression.

4. But the second expression is a **not**, and **not**s always fail on backtracking, so the interpreter returns to the first expression and matches **X=tom** and **Y=dick**.

5. But this **X** now makes the second expression fail--there is a fact that Tom is the boss of Dick.

6. The interpreter returns to the first predicate expression and takes the next matching from the database, **X=ann** and **Y=mary**.

7. This **X** succeeds with the second expression since Ann isn't the boss of Dick.

8. The **Y** match succeeds with the third expression if **Z=harry**. So the interpreter reports that **X=ann, Y=mary, Z=harry**.

Notice the backtracking works differently if we rearrange the query into this equivalent form:

```
?- boss(X,Y), boss(Y,Z), not(boss(X,dick)).
```

This is processed very much like the original two-expression superboss query, except that the **not** expression forces failure instead of success for **X=tom**; we got the same effect by typing a semicolon after the first answer to the query in the last section. But the query won't work right if we reorder it as

```
?- not(boss(X,dick)), boss(X,Y), boss(Y,Z).
```

because now a **not** with an unbound variable is first, violating the guideline of Section 3.6.

The generate-and-test scheme

When a variable occurs more than once in a Prolog query, the Prolog interpreter chooses a value (binding) at the first occurrence, and then uses it in all other occurrences. So processing of the predicate expression containing the first occurrence "generates" a value, which is "tested" by the predicate expressions containing later occurrences. This idea is often used in artificial intelligence, and it's called the *generate-and-test* scheme or paradigm. Often the generating predicate expressions define the types of the variables, so their predicates are type predicates (Section 2.2).

Generate-and-test is a good way to attack problems for which we don't know any particularly good way to proceed. We generate potential solutions, and apply a series of tests to check for a true solution. This works well when it's hard to reason backward about a problem (from a statement of the problem to a solution), but it's easy to reason forward from a guess to a solution (or verify a proposed solution). An example is cryptography (decoding ciphers): an approach is to guess possible coding (encryption) methods, and see if any of them gives coded text resembling a coded message. Many other interesting problems work well for generate-and-test. But problems with well-defined solution methods, like many mathematical problems, aren't suitable for it.

Backtracking with "or"s (*)

Semicolons in queries ("or"s) are tricky for backtracking. We'll mostly ignore them in this book because, as we say, there's a better way to get their effect; but for the record, here's what happens. When a predicate expression before a semicolon succeeds, all the other expressions to the right that are "or"ed with it can be skipped. When such an expression fails, the next term to the right should be tried. If there aren't any more, the whole "or" should fail, which usually means backtracking to the left. So while backtracking with "and"s always goes left, backtracking with "or"s sometimes goes left and sometimes goes right.

Implementation of backtracking

Implementing backtracking requires allocation of a pointer (Appendix C defines pointers) for every predicate expression in a query, a pointer to where in the database the interpreter last found a match for a predicate expression. So Prolog is more complicated to implement than conventional higher-level languages like Pascal that only need extra storage in the form of a stack for procedure calls (Appendix C defines stacks too). Prolog needs a stack for this purpose too, as you'll see in Chapter 4.

Queries don't necessarily require inspection of facts in the database in sequence (*sequential search*). All Prolog interpreters *index* facts in some way, usually at least by predicate name. This means keeping lists of facts with the same predicate name, together with their addresses. So when the interpreter sees an **a_kind_of** predicate in a query, it need only search through the **a_kind_of** facts pointed to in the **a_kind_of** index list for a match. Figure 3-6 gives an example of a database and its index. More selective indexing (not standard in most Prolog dialects, though) can examine the arguments too.

Indexing by predicate name means that Prolog facts can go in many different orders and still provide exactly the same behavior. For instance, the facts in Figure 3-6

```
a_kind_of(enterprise,ship).
a_kind_of(kennedy,ship).
part_of(enterprise,u_s_navy).
part_of(kennedy,u_s_navy).
part_of(u_s_navy,u_s_government).
```

can be rearranged as

```
part_of(enterprise,u_s_navy).
a_kind_of(enterprise,ship).
part_of(kennedy,u_s_navy).
a_kind_of(kennedy,ship).
part_of(u_s_navy,u_s_government).
```

and all queries will give the same answers in the same order, as they will for

```
part_of(enterprise,u_s_navy).
part_of(kennedy,u_s_navy).
part_of(u_s_navy,u_s_government).
a_kind_of(enterprise,ship).
a_kind_of(kennedy,ship).
```

But this is only because the three **part_of** facts maintain their order and the two **a_kind_of** facts maintain their order. The following database will give answers in a different order than the preceding, though it gives the same answers:

```
a_kind_of(kennedy,ship).
part_of(u_s_navy,u_s_government).
part_of(enterprise,u_s_navy).
a_kind_of(enterprise,ship).
part_of(kennedy,u_s_navy).
```

The speed of query answering depends on how many facts are indexed for each predicate in a query; the more facts, the slower the queries. Queries will also be slower when variables appear multiple times in the query and there is no argument indexing. This situation, called a *join* in database systems, requires embedded iterative loops, and loops can take a lot of time. With joins, possibilities literally multiply. For our previous example

```
?- a_kind_of(enterprise,X), color(X,C).
```

if there are 100 **a_kind_of** facts and 50 **color** facts, 50,000 combinations must be tried to find all possible **X** and **C** pairs, as when we type a semicolon repeatedly or when there are no such **X** and **C**.

About long examples

We've studied several long examples in this chapter. Are all the examples of artificial intelligence like this? Yes, unfortunately. Artificial intelligence is a set of techniques for managing complexity, and you can only see its advantages in at least moderately complex problems.

This disturbs some students. They feel that since they can't get all of a long example into their heads at once, they can't really understand what's going on. One reply is to think of programming languages. There's a lot of activity behind the scenes that the programmer isn't usually aware of--parsing, storage management, symbol tables, stacks, type checking, register allocation, and optimization. But you don't need to know this to program. The complexity of artificial-intelligence examples comes from the need to explain, at least initially, similar behind-the-scenes details. Once you understand what details are necessary, you can ignore them as you program.

Most artificial-intelligence programs and systems do provide additional help for understanding complex program activities in the form of *explanation* facilities that summarize and answer questions about reasoning activity. These facilities provide program tracing, and answer questions like "Why did you conclude boss(tom,dick)?" and "Why didn't you conclude boss(dick,tom)?". More on this in Section 15.8.

Keywords:

inference

Prolog database

Prolog interpreter

query

success

failure

variable

binding variables

first-order logic

inputs

outputs

match

matching alternatives

database order

conjunction

match

disjunction

negation

not

built-in predicates

closed-world assumption

consult

backtracking

failure

indexing

generate-and-test

Exercises

3-1. (A) Good programming style in Prolog doesn't allow the argument to a not to be more than one predicate expression, and doesn't allow composite queries (queries not just a single predicate expression) "or"d together.

(a) Using good programming style, write a Prolog query that is true if the "nor" of predicate expressions s a **and b (both of no arguments) is true. ("Nor" means the opposite of an "or").**

(b) Using good programming style, write a Prolog query that is true if the "exclusive or" of predicate expressions a and b is true. ("Exclusive or" means either is true but not both).

3-2. (R,A) Suppose you have the following Prolog database:

```
incumbent(csprofessor,davis).
incumbent(csprofessor,rowe).
incumbent(csprofessor,wu).
incumbent(csprofessor,zyda).
incumbent(cschairman,lum).
incumbent(dean_ips,marshall).
incumbent(provost,shrady).
incumbent(superintendent,shumaker).
incumbent(director_milops,bley).
bossed_by(csprofessor,cschairman).
bossed_by(cschairman,dean_ips).
bossed_by(orchairman,dean_ips).
bossed_by(dean_ips,provost).
bossed_by(provost,superintendent).
bossed_by(director_milops,superintendent).
```

(a) The incumbent predicate means that the person that is its second argument has the job description that is its first argument; the bossed_by predicate means that the boss of the first argument is the second argument. Paraphrase each of the following Prolog queries in English.

```
?- bossed_by(csprofessor,X), bossed_by(X,Y).
?- bossed_by(X,Y), incumbent(X,rowe), incumbent(Y,Z).
?- incumbent(dean_ip,X); incumbent(dean_ips,X).
?- incumbent(J,P), (bossed_by(J,provost); bossed_by(J,dean_ips)).
?- bossed_by(P,superintendent), not(incumbent(P,shrady)).
```

(b) Without using a computer, what will be the first answer found by a Prolog interpreter with the preceding database and with each query given?

3-3. Suppose two queries each represent an "and" of a number of predicate expressions. Suppose the expressions of query 1 are a subset of the expressions in query 2. How do the answers to query 1 relate to the answers to query 2?

3-4. The words "the" and "a" mean different things in English. What important feature of Prolog querying does the difference between them demonstrate in the following sentences?

> **Find a memo we sent headquarters last week. The memo reported on a board meeting last October 10. The board meeting was noisy, and this is mentioned in the memo.**

3-5. (A) Suppose in your Prolog database you have N one-argument facts for the predicate name p and M one-argument facts for the predicate name q.

(a) What is the maximum number of answers, not counting no, that you will get to the query

```
?- p(X), q(Y).
```

(b) How many total times will the Prolog interpreter backtrack from q to p for the situation in part (a), before it types no?

(c) What is the minimum number of answers to the query in part (a)?

(d) What is the maximum number of answers, not counting no, you will get to the query

```
?- p(X), q(X).
```

(e) How many total times will the Prolog interpreter backtrack from q to p for the situation in part (d), before it types no?

(f) What is the minimum number of answers to the query in part (d)?

3-6. (R,A) Suppose we keep in a Prolog database information about grades on two tests in a course.

(a) Suppose we ask if Joe got an A on test 1 and the Prolog interpreter says yes. Suppose we then ask if Joe got an A on test 2 and it says yes. It seems fair to summarize this by saying Joe got A's on both tests 1 and 2. Now suppose we ask if someone got an A on test 1 and the interpreter says yes. We ask if someone got an A on test 2 and it says yes. It is unfair now to conclude that someone got an A on both test 1 and test 2. How is this situation different? How does this illustrate an important feature of Prolog querying?

(b) Suppose the database consists of facts of the form:

```
grade(<person>,<test-number>,<grade>).
```

Write a query that establishes if everyone in the class got an A on test 1, without using an "or" (semicolon). (Hint: Use the exact opposite.)

(c) Suppose you ask if everyone in the class got an A on test 1 and the Prolog interpreter says yes. Suppose you then ask if everyone in the class got an A on test 2 and it says yes. Can you conclude that everyone in the class got both an A on test 1 and an A on test 2? Why? Assume this is a real class at a real college or university.

3-7. Here's a summary of the current situation on the fictitious television soap opera *Edge of Boredom:*

> Jason and Phoebe are married, but Phoebe is in love with Perry. Perry doesn't love her

> because he is still married to Stacey, but Zack is romantically inclined toward Phoebe. He's in competition with Lane, who also loves Phoebe despite being married to Eulalie, whom Jason is feeling romantic about.

(a) Represent the basic meaning of these statements by facts using only two different predicate names. Notice that if X is married to Y, Y is married to X.

(b) A marriage is on the rocks if both its participants are in love with other people and not with each other. Which people are in marriages that are on the rocks? Show the necessary Prolog query and its result.

(c) A person is jealous when a person they love is loved by a third person, or a person is jealous when married to someone loved by a third person. Which people are jealous? Show the necessary Prolog query and its result.

3-8.(a) Consider the query

```
?- a(X,Y), b(X,Y).
```

with the database

```
a(1,1).
a(2,1).
a(3,2).
a(4,4).
b(1,2).
b(1,3).
b(2,3).
b(3,2).
b(4,4).
```

Without using a computer, what are all the answers that you will get to the query, in order (as you keep typing semicolons)?

(b) Without using a computer, what does this query print out (as you keep typing semicolons)?

```
?- a(X,Y), b(X,Y), a(Y,Y).
```

3-9. (A) Consider this Prolog query:

```
?- r(X,Y), s(Y,Z), not(r(Y,X)), not(s(Y,Y)).
```

with this database:

```
r(a,b).
r(a,c).
r(b,a).
r(a,d).
s(b,c).
s(b,d).
s(c,d).
s(c,c).
s(d,e).
```

(a) Without using a computer, what is the first answer found to the query? Hint: you don't have to do it Prolog's way.

(b) Without using a computer, how many times does a Prolog interpreter backtrack from the third to the second predicate expression to get this first answer?

3-10. Consider this Prolog database:

```
u(a,b).
u(b,b).
u(c,d).
u(c,a).
u(d,a).
u(d,c).
```

Now consider this Prolog query, without actually using a computer:

```
?- u(X,Y), u(Y,Z), not(u(X,Z)).
```

(a) How many times will a Prolog interpreter backtrack to the first query predicate expression **u(X,Y)** to find the first answer to this query?

(b) How many times will a Prolog interpreter backtrack to the second query predicate expression **u(Y,Z)** to find the first answer to this query?

(c) How many times will a Prolog interpreter backtrack to the third query predicate expression **not(u(X,Z))** to find the first answer to this query?

(d) How many further times will a Prolog interpreter backtrack to the first query predicate expression

u(X,Y) to find the second answer to this query?

(e) How many further times will a Prolog interpreter backtrack to the second query predicate expression **u(Y,Z)** to find the second answer to this query?

3-11. (H) Design a good set of predicates for the following data about an organization and its employees. Assume you have to do this in Prolog. Try to be efficient: avoid duplicate data, empty data, and too many linking arguments, while keeping data access reasonably fast.

Assume we have an organization with departments, subdepartments, and projects. A subdepartment can belong to only one department, but a project can belong to more than one subdepartment or department (but most of the time only one). Employees belong to one subdepartment and one or more projects. Employees have a name, social security number, date of birth, address, and a list of successfully completed projects that they participated in since they joined the organization. Employees also are characterized by Personnel by "job skills" they have from a rough list (e.g. "can type", "has truck license", "experience in writing"). Projects have a name, code, starting date, projected or actual completion date, and the room number for the office of each employee on the project. Employees have only one office, but there may be more than one employee in the same office.

Design these predicates to answer these questions easily:

> --Give the name, department, and office number for each employee on project 93521.
> --Give the name, department, and office number for each employee on projects started last year.
> --Give the people in department 43 who have typing skills.

3-12. Questions in English have subtleties that are sometimes hard to translate into Prolog queries. This became obvious in building the first *natural language front ends* to databases, computer programs that tried to answer, in English, questions about the database contents. Here are illustrations of two bugs discovered in those early programs. Try to explain what a program like a Prolog interpreter is missing when it makes such errors. (Brackets give our explanatory comments.)

(a)

> Person: Can you tell me the commander of the Enterprise and his rank?
> Computer: Yes. [That's all it types in response.]

(b)

> Person: Who commands the Pequod?
> Computer: Nobody. [That's strange, because every ship must have a commander.]
> Person: Where is the Pequod currently?

Computer: Nowhere. [Strange ship this Pequod.]
Person: Does the Pequod exist?
Computer: No. [So that's the reason.]

Go to book index

Definitions and inferences

Much human intelligence consists of conclusions or *inferences* drawn from facts, instead of facts themselves. We'll now show how inference methods can be represented, including a famous one called *inheritance*. That is, we'll formalize chains of reasoning.

Rules for definitions

If we ask a lot of queries of a database, especially queries about related things, we may be typing the same expressions over and over again. Nearly all programming languages have subroutines or procedures to modularize knowledge, group it into its natural pieces. Prolog has them too, and they're called *rules* or *definitions*. Rules are a way to create the ability to query new predicates without specifying new facts, by defining the new predicate names in terms of old predicate names.

Prolog rules have a *left side* and a *right side*, and the symbol **:-** in between (it is supposed to look like a backward arrow, but you need a good imagination). Read the symbol as "if". To its left side is a single predicate expression, usually with variables as arguments. To its right side is a query, with possibly multiple predicate expressions combined with the comma ("and"), semicolon ("or") and **not** symbols. For instance:

```
gray_enterprise :- part_of(enterprise,X), color(X,gray).
```

This says that predicate expression **gray_enterprise** succeeds if the query

```
?- part_of(enterprise,X), color(X,gray).
```

succeeds. In other words, **gray_enterprise** is true if the Enterprise is part of something that is gray in color. So **gray_enterprise** is a code representing a query.

More formally, the **gray_enterprise** rule defines a new predicate of no arguments called **gray_enterprise**. When queried:

```
?- gray_enterprise.
```

the interpreter succeeds (answers **yes**) if it can succeed in querying the right side of the **gray_enterprise** rule. Otherwise the interpreter fails (answers **no**). It's like the right side of the rule is substituted for the left side whenever it occurs in a query. The variable **X** is a *local variable* in the rule, a variable whose value will be "thrown away" when the rule is done, and will not be printed out in any query answer. Any Prolog variable that appears only on the right side of a rule is called local to that rule.

But **gray_enterprise** isn't too useful because it only covers one color. What we really want is a rule that will *tell us* what color something is. We can do this with a variable on the left side of the rule, a variable representing the color:

```
color_enterprise(C) :- part_of(enterprise,X), color(X,C).
```

C is a *parameter variable* of the rule, and a value for it will be "returned" as a result of executing the rule. So if we query

```
?- color_enterprise(C).
```

when our database is that of Section 3.1 and the interpreter uses the **color_enterprise** rule, it will type

```
C=gray
```

in reply. Rules can have any number of such parameter variables, as well as constants (ordinary words and numbers), as arguments on their left sides. So if we wanted an even more general color rule, one that would tell us the color for many objects, we could say:

```
color_object(X,C) :- part_of(X,Y), color(Y,C).
```

Here **X** and **C** are parameter variables, and **Y** is a local variable. So values for **X** and **C** only will be returned.

Figure 4-1 summarizes our terminology about rules. You can think of local variables as being existentially quantified ("there exists some X such that something is true"), and parameter variables as universally quantified ("for every X something is true")--see Appendix A for more about quantification, an important concept in logic. Actually, the distinction of "local" from "parameter" variables is misleading, because the opposite of "local" is "global", and there aren't any true "global" variables in Prolog. Rules just give a shorthand for queries, and a variable **X** is one query is always different from a variable **X** in a separate query. The only way to get anything like a global variable in Prolog is to assert a fact.

Here are more rule examples. This defines X to be the son of Y if X is the child of Y and X is male:

```
son(X,Y) :- child(X,Y), male(X).
```

This defines X to be the "superboss" of Y if X is the boss of the boss of Y:

```
superboss(X,Y) :- boss(X,Z), boss(Z,Y).
```

This defines something X you own to be stolen if it isn't present and you haven't given it to someone:

```
stolen(X) :- owns(you,X), not(present(X)), not(given(you,X,Y)).
```

This defines the previous example predicate **color_object** in a different way, using **a_kind_of** instead of **part_of**. It says that the color of X is C if X is a type of Y and the color of Y is C.

```
color_object(X,C) :- a_kind_of(X,Y), color(Y,C).
```

Usually predicates defined by rules can be used in queries just like fact predicates. This means that we can usually make variables any of the arguments to defined-predicate expressions in queries, a powerful feature. (A few exceptions to this will be discussed in later chapters: some rules with **not**s, the arithmetic and **is** constructs of Chapter 5, and the "cut" ("**!**") of Chapter 10.) That is, we can designate arbitrary inputs and outputs. This means that Prolog is fundamentally more powerful than conventional programming languages, which usually require all but one argument (the output variable) to a procedure to be filled in at the time of the call. But Prolog rules, the Prolog equivalent of procedures, rarely impose such requirements.

Rule and fact order

Rules can go into a Prolog database just like facts. And the interpreter then uses both those rules and facts to answer queries. But we must worry a little about rule and fact order.

To see why, consider combining several reasoning methods in the same rule, as for instance three ways of establishing the color of something:

```
color_object(X,C) :- color(X,C); (part_of(X,Y), color(Y,C));
(part_of(X,Y), part_of(Y,Z), color(Z,C)).
```

(Queries and rules can take more than one line; the period at the end indicates where they stop.) But that's poor rule-writing style, since it's hard to read the right side. Instead, we can write three separate rules with the same left side:

```
color_object(X,C) :- color(X,C).
color_object(X,C) :- part_of(X,Y), color(Y,C).
color_object(X,C) :- part_of(X,Y), part_of(Y,Z), color(Z,C).
```

Now each rule's right side gives *sufficient* conditions for the left-side predicate expression to be true, but not *necessary* conditions. That is, each describes some but not all the situations for which **color_object** succeeds. When each rule is very specific, we have *definition by examples*.

The order of those three rules matters. Whenever a query on predicate **color_object** with two arguments is issued, the rules will be tried to answer the query in database order, just as if they were facts. The order shown is probably the best, because the simplest rule will be tried first, then the next-simplest, then the most complex (where we measure "complexity" by the number of expressions).

Facts and rules can be freely intermingled in a Prolog database; the overall order is all that matters. Facts should generally come first, though, if there are both facts and rules for a predicate name. That's because facts require less symbol matching than rules--and a lot less than some of the recursive rules we'll discuss later in this chapter. Putting facts before recursive rules of the same predicate is also the standard way to write recursive programs (see Appendix B).

Rules as programs

So this is how we'll write a Prolog program: by giving a list of facts and rules (definitions) that explain about some area of human knowledge. To match a query, rules and facts will be considered in order, somewhat like sequential execution of the lines of a program in a conventional computer language. "And"s on the right sides of queries will be initially be done in left-to-right order, like multiple procedure calls in a line of a conventional language. And like those languages, we can have "procedure" (rule) hierarchies; that is, a rule right side can use predicate names defined in rules, including its own left-side predicate name (the last is called *recursion*).

This mapping of predicate expressions to actions means that we can model *procedures* in Prolog, not just facts. For instance, here's a way to describe a daily agenda for a student:

```
day_agenda :- wakeup, classes, lunch, classes, dinner, study, sleep.
wakeup :- late, dress, travel(home,campus).
wakeup :- not(late), shower, dress, breakfast, travel(home,campus).
classes :- class, class, class.
class :- check_schedule, go_room, take_notes.
```

This notation is useful for describing processes, even if we never define what the basic actions (like **take_notes**) are. A computer doesn't need to know *everything* to be intelligent, just the important things. (However, we'll introduce a better way to describe sequences of actions in Chapter 9.)

Rules in natural language

As we said in Section 2.8, an artificial-intelligence system is often built from natural-language (e.g. English) specifications, either oral or written. Rules can be specified several ways in English. The easiest to spot is the "if...then" statement:

> If a vehicle floats on water, then it's a ship.

which could become the Prolog rule (note the reversed order):

```
ship(X) :- vehicle(X), floats(X,water).
```

But the "then" is often omitted:

> If a vehicle floats on water, it's a ship.

"Define" or "assume" often signals a rule, taking the things in the opposite (Prolog) order:

> Define a ship as anything that floats on water.
> Assume as a ship anything that floats on water.

"Anything", "anyone", "any", "something", "some", and "a" in such definitions often map to Prolog variables.

Besides expressing facts, the verb "to be" can express rules:

> A ship is any vehicle that floats on water.
> Ships are water-floating vehicles.

The borderline between facts and rules can be fuzzy, but generally speaking, use a rule when variable bindings seem possible. The "a" and "any" in the first sentence suggest a variable, and hence a rule.

When an "and" occurs in the "if" part of a definition, we can just put commas into the right side of the rule. For instance

> Something that is a vehicle and floats on water is a ship.

takes the form of the preceding **ship(X)** rule. If an "and" occurs in the "then" part of a definition, multiple conclusions hold for some situation. Prolog doesn't allow "and"s in rule left sides, but we can write multiple rules with the same right side and different left sides. So

> If a vehicle floats on water and is gray, then it is a ship and of military origin.

becomes

```
ship(X) :- vehicle(X), floats(X,water), gray(X).
origin(X,military) :- vehicle(X), floats(X,water), gray(X).
```

Rules without right sides

The right side of a rule represents sufficient conditions for the left-side predicate expression to be true. What if a rule doesn't have anything on its right side? Then we'd be saying that nothing is necessary to make the left side true, that the left side always true. In other words, a fact. So facts are just a special case of rules. That's why Prolog facts and rules are put together in one big database: they're the same thing.

But rules can have variables. What would it mean for a fact to have a variable? Consider:

```
part_of(X,universe).
```

If you think of that as a rule with no right side, it says that for any **X**, nothing is necessary to say that **X** is part of the universe. In other words, every **X** is part of the universe. So using a term from logic (see Appendix A), facts with variables are *universally quantified*.

Postponed binding

An interesting consequence of the Prolog interpreter's handling of rules is "postponed" binding of variables. This is interesting because most other programming languages can't do anything like this. Suppose we query:

```
?- color_object(enterprise,C).
```

Here the first argument is bound (that is, it's an input) and the second argument is unbound (that is, it's an output). If there's a fact in the database

```
color_object(enterprise,gray).
```

then the query can immediately bind **C** to **gray**. But if there are no **color_object** facts, only rules with **color_object** on their left sides, or if a rule is first in the database, the binding may be delayed. For instance, suppose the interpreter picks the rule

```
color_object(X,C) :- part_of(X,Y), color_object(Y,C).
```

The variable **C** won't be bound when the rule is invoked, and it won't be bound in the **part_of** (which doesn't mention **C**), so if it's ever bound it won't be until the recursive query of **color_object**. But this query may require other recursive calls. It may take a long time for an appropriate **color_object** fact to be found, and hence a long time for variable **C** to get bound.

In fact, some query variables may *never* get bound to values in successful queries. Consider:

```
recommended(X,J) :- not(bad_report(X)).
```

a rule that recommends person **X** for job **J** if they haven't had a bad report recently. Here **J** is never bound if the query's second argument isn't initially bound. Most Prolog dialects will invent a name like "**_14**" for such unbound variables, to print out if they must.

Postponing of binding in Prolog interpreters has the advantage that binding is only done when truly necessary to answer a query. This saves on the computer's overhead cost of binding, for one thing. Furthermore, *multiway* reasoning becomes easier, reasoning for which we don't know in advance which arguments to a predicate expression (procedure) will be bound (or be inputs). Multiway reasoning means the same rule can be used many ways. We'll look more into this in the next chapter.

Backtracking with rules

As you remember from Chapter 3, backtracking is the "backing up" the interpreter does when it can't find a match for a predicate expression in a query. When predicates in a query have rule definitions, the interpreter acts somewhat as if the right side of the definition were typed as part of a query instead of the single predicate expression that is the left side. (We say "somewhat" because variables in a rule are local in meaning, and parameter variables must get bound.) So backtracking into a predicate expression for a rule-defined predicate means returning to the last expression in the rule. Tracing backtracking with rules is hard because we move both left and right (through backtracking) and up and down (through procedure calls and returns). This is a common problem with powerful computer languages such as Prolog: simply because they're powerful, it's hard to follow everything they're doing.

Here's an example that isn't too hard. Suppose we have two kinds of facts about an organization: the department each employee works for and the manager of each department. Suppose we define a **boss** predicate of two parameter arguments **B** and **E**, that says that **B** is the boss of **E** if **B** manages a department **D** in which **E** is an employee; **D** will be a local variable. Assume that Tom works in the sales department, Harry works in the production department, Dick manages the sales department, and Mary manages the production department. Then the Prolog database is:

```
department(tom,sales).
department(harry,production).
manager(dick,sales).
manager(mary,production).
boss(B,E) :- department(E,D), manager(B,D).
```

Now suppose we want to find a boss different from Tom's. It will be **X** in the query:

```
?- boss(X,Y), not(boss(X,tom)).
```

Notice that the **not** must come second in the query because **X** must be bound.

Let us trace execution (see Figure 4-2).

1. The first predicate expression in the query matches only a rule in the database, no facts, so the first job to do is searching for a match for the first expression on the right side of the rule, **department(E,D)**. This can be matched to the first fact, with **E=tom** and **D=sales**.

2. Moving on to the second predicate expression in the rule, **manager(B,D)**, the interpreter finds a match in the third fact with **B=dick**, so the rule succeeds and the first expression in the original query succeeds. So **X=dick** and **Y=tom**.

3. Moving to the second and last expression in the query, **not(boss(X,tom))**, the interpreter tries to find if Dick is the boss of Tom. It has no memory of what it just proved, so it goes back to the rule.

4. Again, both predicate expressions on the right side of the rule can match the same facts, so the rule succeeds. But the condition in the original query is the opposite ("not" or negation) of this, so the second half of the original query fails and the interpreter backtracks to the first expression. (There's never a new way to satisfy a **not** when it fails.)

5. Backtracking into a predicate expression satisfied previously by a rule means that the interpreter must go to the last (rightmost) expression on the right side of the rule and see if it can find another match there. But there is no other boss of the sales department so it must now backtrack to the first expression of the rule, **department(E,D)** with unbound **B** and **E**.

6. Fortunately for this there is another choice: **E=harry** and **D=production**.

7. With this success, the interpreter can start moving right again. It considers the second predicate expression in the rule. And it can find a match of **B=mary** (remember, **D=production** now). So the rule succeeds, and thus the first expression of the query succeeds with **X=mary** and **Y=harry**.

8. Now in the second query expression, it must check that it is not true that Mary is the boss of Tom. To do this, it tries to prove the **boss** rule fails with **B=mary** and second argument **tom**.

9. In the first expression on the right side of the rule, it can match **D** to sales, but there is no fact that Mary manages the sales department. So the rule fails, and since there is no other rule or fact for the **boss** predicate, the second **boss** expression in the query fails.

10. But since there's a **not** in front of this expression, the whole query succeeds. So

X=mary is the answer we needed.

To help in debugging, most Prolog interpreters will automatically print out abbreviated trace information if you ask. To ask in most Prolog dialects, query the built-in predicate **trace** of no arguments. To stop the tracing, query the built-in predicate **notrace** of no arguments. Some other debugging facilities of most Prolog dialects are described in Appendix D.

Transitivity inferences

Certain rule forms occur frequently in artificial intelligence. A very important form states *transitivity* of a two-argument predicate. For instance, consider bosses in an organization. If your boss has a boss in turn, that big boss is your boss too. If that big boss has a boss in turn, that even bigger boss is your boss too. So bossing relationships form chains, and that's transitivity.

Formally, a relationship predicate **r** is transitive if this rule is correct:

```
r(X,Y) :- r(X,Z), r(Z,Y).
```

(See Figure 4-3.) This says that if predicate **r** holds from some **X** to some **Z**, and also from **Z** to some **Y**, the predicate always also holds from **X** to **Y**. This rule can be used recursively too; that is, it can refer to itself on its right side, not just to **r** facts. So the rule can follow indefinitely relationship long chains. For instance, suppose these facts are placed in front of the rule in the Prolog database:

```
r(a,b).
r(b,c).
r(c,d).
```

Then if we query

```
?- r(a,d).
```

no facts match, so the interpreter will use the rule, and will first query

```
?- r(a,Z).
```

For this, **Z** can match **b** in the first fact. The interpreter will then query

```
?- r(b,d).
```

There's no fact stating this either, so it must use the rule again recursively. For this new call of the rule, **X=b** and **Y=d**, so the next query is

```
?- r(b,Z).
```

This new **Z** is different from the previous **Z**, since each recursive call has its own variables (see Appendix B), in the same way that **X**'s in different rules represent different **X**'s. For this new query, the interpreter can bind **Z** to **c** since there is a **r(b,c)** fact, and then the second part of the rule becomes the query:

```
?- r(c,d).
```

That predicate expression is a fact. So the rule succeeds in proving **r(b,d)**. And thus it succeeds in proving **r(a,d)**, the original query.

Many of the relationship predicates in Section 2.6 are transitive: **a_kind_of**, **part_of**, **right_of**, **during**, and **ancestor**, for instance. Some example applications:

> --If the Vinson is a kind of carrier, and the carrier is a kind of ship, then the Vinson is a kind of ship.
>
> --If the electrical system is part of the car, and the battery is part of the electrical system, then the battery is part of the car.
>
> --If the Vinson is north of the Enterprise, and the Enterprise is north of the Kennedy, then the Vinson is north of the Kennedy.
>
> --If during the day Monday you had a meeting with your boss, and during the meeting you found out you got promoted, then during the day Monday you found out you got promoted.
>
> --If a number X is greater than a number Y, and Y is greater than a number Z, then X is greater than Z.

And a graphical example is shown in Figure 4-4. Here we are representing facts about a pile of blocks on a table. A block is above another block if it is resting on that block. Block b is above block a, block c is above block b, and block d is above block c. Hence by transitivity, block d is above block a.

Why is transitivity important? Because it can save fact space, by figuring out "indirect" facts from a few direct ones. So it reduces redundancy. Transitivity will work best when we store facts relating only the "closest" things--like for **part_of**, the very smallest thing B that still contains thing A. That's because transitivity explains relationships between things farther apart from the same relationships between things closer together, not the other way around.

Inheritance inferences

An even more important rule form for artificial intelligence is the *inheritance* form. Consider a bureaucratic organization. If it has only one business address, then that is the business address of all the employees. It wouldn't make sense for a computer to keep a separate business address fact for each employee; that would mean a lot of unnecessary facts. Instead it should store a single address fact with the name of the organization, and reason from that. This reasoning is called inheritance; we say the address *inherits* from organization to employee.

Inheritance always involves two predicates, a property predicate and a relationship predicate. Formally, we say property predicate **p** inherits with respect to relationship predicate **r** if this rule is correct:

```
p(X,Value) :- r(X,Y), p(Y,Value).
```

(See Figure 4-5.) That is, we can prove that property **p** of **X** has value Value if we can prove the **Y** is related to **X** by predicate **r**, and **Y** does have value Value for property **p**. (This generalizes the rule for predicate **color_object** in Section 4.1.) Sometimes we use the term "inheritance" when **p** is a relationship predicate too. Like the transitivity rule, the inheritance rule can be used recursively--that is, the **p** on the right side can use the rule itself to achieve its ends--but this isn't too common because the **r** predicate can recurse too.

Inheritance frequently occurs when the relationship predicate (**r** in the preceding) is **a_kind_of**. That is, if you want to know some property **p** of some **X**, find some **Y** that Xis an example of, and **Y**'s value is **X**'s value too. Some examples:

> --If people are animals, and animals eat food, then people eat food.
>
> --If the Enterprise is a kind of ship, and ships float on water, then the Enterprise floats on water.

But inheritance can occur with relationship predicates besides **a_kind_of**:

> --If the hood is part of my car's body, and my car's body is gray, then the hood is gray too.
>
> --If the U.S. Navy is part of the U.S. Government, and U.S. Government is everywhere mired in bureaucratic inefficiency, then the U.S. Navy is everywhere mired in bureaucratic inefficiency.
>
> --If the Enterprise is at Norfolk, and Captain Kirk is on the Enterprise, then Captain Kirk is at Norfolk.

And a semantic network example is shown in Figure 4-6. Here we have facts about a particular truck called **truck_4359** and its battery. The truck has a location (**nps_north_lot**) and an owner (**nps**). These properties inherit *downward* to the battery via the **part_of** relationship to the battery, and that's what those two dotted lines from the battery node mean. ("Downward" means from a larger or more general thing to a smaller or more specific thing.) In other words, the location of the battery of a truck comes from the location of the whole truck, and the owner of the battery comes from the owner of the whole truck. Inheritance can also proceed upward from the battery. The status property of the battery (**defective**) inherits upward, because we can say that the truck is defective when its battery is defective.

Generally, properties that represent universal or "for every X" quantification (see Appendix A) inherit downward, while properties that represent existential or "there exists an X" quantification inherit upward. For instance, the "animals eat food" example is downward: this says that for every X an animal, it eats food, and so since people are animals, they eat food too. But consider the property of people that some of them live in North America, i.e. that there exist some people that live in North America. Then since people are animals, some animals live in North America, so that existential property inherits upward.

Inheritance of a property need not be absolute--there can always be exceptions. For instance, the battery may be removed from the truck for repair, so even though the truck location is the NPS north lot, the battery location isn't. For such an exception, the location of the battery can be put into the database in front of the inheritance rule, so a query on the truck's battery location will find the fact first. So an inheritance rule is a *default*, general-purpose advice that can be overridden.

Inheritance is often associated with transitivity, because when the relationship predicate **r** is transitive, the interpreter is actually following a transitivity chain to follow an inheritance chain. For instance (see Figure 4-7), suppose we have the facts

```
a_kind_of(vinson,carrier).
a_kind_of(carrier,ship).
a_kind_of(ship,vehicle).
purpose(vehicle,transportation).
```

and a rule for inheritance of **purpose** with respect to **a_kind_of**:

```
purpose(X,V) :- a_kind_of(X,Y), purpose(Y,V).
```

and a rule for transitivity of **a_kind_of**:

```
a_kind_of(X,Y) :- a_kind_of(X,Z), a_kind_of(Z,Y).
```

Suppose we query:

```
?- purpose(vehicle,P).
```

We can use the transitivity rule to prove first **a_kind_of(vinson,ship)**, then **a_kind_of(vinson,vehicle)**; the latter inferred relationship is shown by a dotted link in the Figure. Then the inheritance rule succeeds, proving **purpose(vinson,transportation)**; this is also shown by a dotted link.

Thus inheritance is useful for the same reason as transitivity: it can extrapolate an explicit set of facts to a much larger implicit set of facts, by inferences with rules. It usually makes sense, then, to only store "nonredundant" property facts that can't be inherited (though it won't give wrong answers to store redundant information, just increase the size of the database and slow other queries a little). So we typically should store property facts about properties for the *most general* things having those properties. This is because inheritance usually goes downward from more general things to more specific things.

Some implementation problems for transitivity and inheritance

Unfortunately, peculiarities of Prolog interpreters can prevent transitivity and inheritance rules from working properly. The problem is a possible *infinite loop*, a frequent danger in Prolog programming. An infinite loop is when the same thing is done over and over forever when trying to answer a query. Typically, a rule calls itself forever or a set of rules call one another in cycle forever. (A good clue is the error message "Out of stack space" when running a simple program.) You can get into infinite loops in other programming languages too, but it's easier with Prolog because of its emphasis on recursion and complicated backtracking.

To avoid infinite loops for inheritance, we must pay careful attention to database order. We must put any rules about inheriting values for some property after facts about values of that property. We should also use

```
p(X,Value) :- r(X,Y), p(Y,Value).
```

instead of

```
p(X,Value) :- p(Y,Value), r(X,Y).
```

The second rule can cause an infinite loop in which **p** continually calls itself with the same arguments. This won't happen at first, if facts that can match the **p** query come before this rule in the Prolog database; but whenever the proper answer is **no** (either when the only possible answer to the query is **no**, or when we type enough semicolons after other answers), the interpreter will recurse forever instead of saying **no**. This is because Prolog executes *top-down* like other programming languages: the left side of a rule is matched to some query, and then the right side provides new things to match. So be careful with inheritance rules. In fact, as much British detective fiction demonstrates, inheritances can lead to murder.

But it's tougher to eliminate the infinite loop with the transitivity rule:

```
r(X,Y) :- r(X,Z), r(Z,Y).
```

This works fine when the **r** relationship is provable from the facts, but will never say **no** when **r** is unprovable. For instance, query

```
?- r(a,b).
```

won't say **no** when there are no facts in the database involving either **a** or **b**: the first predicate expression on the right side has no choice but to call on this rule itself. Here reordering the expressions on the right side of the rule won't change anything, because both have predicate **r**. Instead we must rewrite the rule as two, and use another predicate name, one for queries only. For instance, consider transitivity for the **boss** relationship, where **boss(X,Y)** means that person **X** is the boss of person **Y**. To describe indirect bosses, we'll use a new predicate **superior** of two arguments:

```
superior(X,Y) :- boss(X,Y).
superior(X,Y) :- boss(X,Z), superior(Z,Y).
```

This will avoid the infinite loop because the only recursion is in the second rule, and the form of the second rule is the same as the first (better) form for inheritance rules given previously.

A similar trick can be used to state symmetry or *commutativity* of a two-argument predicate. Examples of commutative predicates are the **equals** predicate for numbers, the "distance-between" predicate for places, and the "is-friends-with" predicate for people. The obvious rule can easily cause infinite loops:

```
p(X,Y) :- p(Y,X).
```

Querying a **p2** instead avoids this problem:

```
p2(X,Y) :- p(X,Y).
p2(X,Y) :- p(Y,X).
```

provided we don't use **p2** in a recursive rule itself.

A longer example: some traffic laws

Rules can do many things. As we have seen, rules can define new predicates and extend the power of old predicates. Rules can also state policies, prescriptions of what to do in particular situations. Here is an example of the representation of California traffic laws. These are the laws about signal lights, for both vehicles and pedestrians, from the *California Driver's Handbook*, 1985. The letters in brackets are paragraph codes for later reference.

> [A] New Signals-Note: California is adopting red arrows and yellow arrows in addition to

> green arrows, as signals for vehicle traffic. This is what the colors of traffic lights mean: A red light or arrow means "STOP" until the green appears. A flashing RED traffic light or arrow means exactly the same as a stop sign, namely STOP! But after stopping, proceed when safe, observing the right-of-way rules.
>
> [B] A GREEN light or arrow means "GO", but you must let any vehicles or pedestrians remaining in the intersection when the signal changes to green, get through before you move ahead. Look to be sure that all cross traffic has stopped before you enter the intersection.
>
> [C] A YELLOW light or arrow warns you that the red signal is about to appear. When you see the yellow light or arrow, you should stop if you can do so safely. If you can't stop, look out for vehicles that may enter the intersection when the light changes. A flashing YELLOW light or arrow is a warning of a hazard. Slow down and be especially alert.
>
> [D] A lighted GREEN ARROW, by itself or with a RED, GREEN or YELLOW light, means you may make the turn indicated by the green arrow. But give the right-of-way to pedestrians and vehicles which are moving as permitted by the lights. The green arrow pointing left allows you to make a "protected" left turn; oncoming traffic is stopped by a red light as long as the green arrow is lighted.
>
> [E] If the green arrow is replaced by a flashing yellow light or arrow, slow down and use caution; make the move which the green arrow would permit, only when safe for you and others.
>
> [F] If the green arrow is replaced by a flashing red light or arrow, stop for either signal; then go ahead when it's safe to do so.
>
> [G] NEW SIGNALS--Note: California is adopting international symbols to guide pedestrians at street crossings. An upraised hand (orange) means the same as the "WAIT" or "DON'T WALK" sign. A walking person symbol (white) means the same as the "WALK" sign.
>
> [H] Special signs for walkers: The "DON'T WALK" or "WAIT" or upraised hand sign, if flashing, warns the walker that it is safe to cross, first yielding to any vehicles which were still in the crossing when the light changed.
>
> [I] At a crossing where there are no special pedestrian signals, walkers must obey the red, yellow, or green lights or arrows. But walkers facing a green arrow must not cross the street.

We can represent the preceding text as a set of Prolog rules in four groups: rules about arrows, rules about cars, rules about pedestrians, and *default* rules (the last two **action** rules). As we said, defaults are weak prescriptions for general cases, only used if more specific advice cannot be found; for instance, the first default rule says that if nothing prevents you from going forward, go forward.

In listing these rules, we follow a standard Prolog convention of putting blank lines between groups of rules with the same left-side predicate name; this makes reading long programs easier. We also use comments; anything between the symbols "/*" and "*/" is treated as a comment and ignored by the Prolog interpreter. The codes at the beginning of lines reference the relevant paragraph(s) of the *Driver's Handbook* text. Note that the text and rules do not correspond exactly; in fact, some "obvious" rules included are nowhere in the text. What's "obvious" to people isn't always so obvious to computers.

```
/* ------ Rules for arrow lights ------ */
/* A */ action(car,stop) :- light(yellow_arrow,Direction),
  safe_stop_possible.
/* D,E */ action(car,yield_and_leftturn) :-
  light(yellow_arrow,left), not(safe_stop_possible).
/* D,E */ action(car,yield_and_rightturn) :-
  light(yellow_arrow,right), not(safe_stop_possible).
/* D */ action(car,yield_and_leftturn) :- light(green_arrow,left).
/* D */ action(car,yield_and_rightturn) :- light(green_arrow,right).
/* ------ Rules for regular lights ------ */
/* A */ action(car,stop) :- light(red,steady).
/* A */ action(car,stop_and_go) :- light(red,flashing).
/* C */ action(car,stop) :- light(yellow,steady),
  safe_stop_possible.
/* C */ action(car,yield_and_go) :- light(yellow,steady),
  not(safe_stop_possible).
/* B */ action(car,yield_and_go) :- light(green,steady).
/* C */ action(car,slow) :- light(yellow,flashing).
/* A */ action(car,stop) :- light(red_arrow,Direction).
/* ------ Rules for pedestrian lights ------ */
action(pedestrian,stop) :- pedhalt(steady).
/* I */ action(pedestrian,stop) :- not(pedsignals), greenfacing.
action(pedestrian,stop) :- pedhalt(flashing),
  safe_stop_possible.
/* H */ action(pedestrian,yield_and_go) :- pedhalt(flashing),
  not(safe_stop_possible).
/* H */ action(pedestrian,yield_and_go) :- pedgo(steady).
/* ------ Default rules ------ */
/* I */ action(pedestrian,A) :- not(pedsignals), not(greenfacing),
action(car,A).
action(X,go) :- not(action(X,stop)),
  not(action(X,stop_and_go)).
```

```
/* ------ Rules defining the special terms ------ */
/* G */ pedhalt(State) :- light(wait,State).
/* G */ pedhalt(State) :- light(dont_walk,State).
/* G */ pedhalt(State) :- light(hand,State).


/* G */ pedgo(State) :- light(walk,State).
/* G */ pedgo(State) :- light(walking_person,State).


pedsignals :- pedhalt(State).
pedsignals :- pedgo(State).


greenfacing :- light(green_arrow,right),
  not(clockwise_cross).
greenfacing :- light(green_arrow,left),
  clockwise_cross.
```

To understand programs like this, it's often a good idea to draw a *predicate hierarchy* (actually, a lattice) showing which predicates refer to which other predicates within their definitions. This helps distinguish *high-level* predicates about abstract things from *low-level* predicates about details. Figure 4-8 shows the predicate hierarchy for this program.

The rules of this program define legal actions in particular traffic situations. To use them, we query the predicate **action** with unbound variables (outputs) for one or both of its arguments. The first argument to **action** represents whether you are a pedestrian or in a car, and the second argument represents a legal action in the situation. So for instance, **action(car,stop)** means that it is legal for the car to stop. The most useful way to query **action** is to make the first argument an input and the second argument an output. So to know what is legal for a car, query

```
?- action(car,X).
```

and the variable **X** will be bound to a description of some action legal with the current facts; to know what is legal for a pedestrian, query

```
?- action(pedestrian,X).
```

To check if some particular action is legal, fill in both arguments. If a sequence of actions is legal, instead of just one, the program puts the characters "**_and_**" between the actions to form one long word.

To describe a situation, three kinds of facts are needed:

--**light(<kind-of-light>,<light-status>)**: this is the main kind of fact. The first argument is either:

red (a circular red light)
yellow (a circular yellow light)
green (a circular green light)
red_arrow
yellow_arrow
green_arrow
wait (the word "wait" lit up)
dont_walk (the words "don't walk" lit up)
hand (a picture of a human palm)
walk (the word "walk" lit up)
walking_person (a picture of a walking person)

The second argument describes the condition of the light; this is either **left** or **right** for the arrow lights, and either **steady** or **flashing** for all the other lights.

--**safe_stop_possible**: this predicate of no arguments asserts that you can stop quickly and safely.

--**clockwise_cross**: if you are a pedestrian, this predicate of no arguments asserts that the path by which you will cross a street is clockwise with respect to the center of that street.

For example, here are the facts for when you are in a car approaching an intersection with plenty of room to stop, and you see a steady yellow light and a green arrow light pointing to the left:

```
safe_stop_possible.
light(yellow,steady).
light(green_arrow,left).
```

Rule order matters, since it establishes priorities among actions. Rules for arrows should go before rules for other kinds of lights, because arrows override other lights showing. Within each of the three main groups of rules--arrows, regular lights, and pedestrian lights--the rules for stopping should go first to handle malfunctions of the lights in which more than one light is lit. Finally, defaults should go last, like the last two **action** rules.

Running the traffic lights program

Here are some examples of the program working. Suppose our database is the example facts just given.

Suppose we ask what a car can do:

```
?- action(car,X).
```

The first three rules fail because there are no facts about yellow arrows. But a green arrow to the left is visible, so the fourth rule succeeds, and we get

```
X=yield_and_leftturn
```

In other words, in this situation it is legal to yield and then make a left turn. Now if we type a semicolon, the interpreter will try to find an alternative action. It starts from where it left off in the rules, so it next examines the fifth rule. Neither the fifth, sixth or seventh rules are satisfied by the three facts, but the eighth rule does succeed. So we get

```
X=stop
```

In other words it is legal to stop your car. If we type a semicolon once more, the query will fail (the last rule fails because it is legal to stop the car, as we just showed). So there are only two alternatives.

For another example, suppose a pedestrian walking along a sidewalk comes to an intersection where they wish to cross the street in a clockwise direction with respect to the center of the intersection. Suppose that in the direction the pedestrian wants to go there is a steady green light and a flashing white picture of a walking person. The representation is:

```
clockwise_cross.
light(green,steady).
light(walking_person,flashing).
```

To discover what we can do, the query is

```
?- action(pedestrian,X)
```

None of the first twelve rules succeed because this query specifies a pedestrian. For the thirteenth rule to succeed, a **pedhalt** rules must succeed, but the lights we observe are not any of the three **pedhalt** types. So the thirteenth rule fails, and similarly the fourteenth and fifteenth. For the sixteenth rule we must check the **pedgo** rules. The first **pedgo** rule fails, but the second matches our last fact with the **State=flashing**. So the sixteenth rule succeeds, and we get typed out

```
X=yield_and_go
```

which recommends to the pedestrian to first yield the right-of-way, then go ahead.

This program should be invoked repeatedly, say every second to decide what to do for that second. And each second something must delete old facts and write new facts into the Prolog database. This general approach is followed by most real-time artificial intelligence programs. For speed, each updating can be done by a separate concurrent processor, to avoid interfering with inference. Notice that we have ignored the biggest obstacle to making this program practical, the recognizing of lights in the real world. This is addressed in the "vision" subarea of artificial intelligence.

Declarative programming

The traffic lights program may not seem much like programs in other programming languages. Programming in Prolog (and much artificial intelligence programming, whatever the language) is different in style from most programming. The style is programs in lots of small modular pieces, pieces smaller than the usual subroutines and procedures of other languages. The emphasis is on writing correct pieces, and not on putting the pieces together. In writing and debugging each piece, the emphasis is on whether it makes sense by itself and whether it is logically correct, not how it is used--the "what" instead of the "how". This is called *declarative* programming | REFERENCE 1|. .FS | REFERENCE 1| It's also close to what is called *nondeterminism* or *nondeterministic programming*. .FE

This bothers some students. They feel they don't really understand artificial intelligence programs because there's often no clear, easily understandable sequence in which programs do things. Prolog interpreters for instance work from top to bottom through facts and rules with the same predicate name, but for each new predicate in a query, they jump to the first occurrence of that predicate name, and jump around in quite complicated ways when they backtrack. And Prolog's operation represents one of the simpler ways to do artificial intelligence programs, as we'll see in Chapter 6. A programmer accustomed to flowcharts may find this bewildering.

In reply, one can say that artificial intelligence solves hard problems, problems on which conventional software-engineering techniques (including numerical ones) struggle, and for which artificial-intelligence methods seem to be the only ones that work. And there are no clear criteria for "intelligent behavior", so in designing intelligent programs it would seem more important to ensure that the individual pieces of the program make sense rather than imposing some grand (necessarily controversial) organizing scheme. Usually, researchers don't want artificial intelligence programs to be brilliant, just not dumb, and concentrating on the pieces helps avoid being dumb. Also, the well-known software technique of recursion (see Appendix B and Chapter 5) is best understood in a declarative way, and artificial intelligence often uses recursion.

This still may not sound too reassuring, but remember there are lots of ways to program. As we say in California, stay mellow.

Keywords:

rule
left side of a rule
right side of a rule
local variable in a rule
parameter variable of a rule
hierarchies of rules
postponed binding
transitivity of a predicate
inheritance of a predicate
commutativity of a predicate
infinite loop
database order
default
declarative programming

.SH Exercises

4-1. (E) What's the difference between "**:-**" and logical implication? In other words, what's the difference between **a :- b.** and the logical statement

> "If b is true, then a is true."

4-2. (A) (a) Suppose in a database you had a rule whose right side was guaranteed to be always true. How could you get the effect of such a rule in a simpler way?

(b) Suppose in a database you had a rule whose right side was always guaranteed to be false for any assignment of bindings. How could you get the effect of such a rule in a simpler way?

4-3. Write rules for reasoning about genealogies (or "family trees"). Assume the genealogy is represented by facts of the single four-argument form

```
child(<name_of_father>,<name_of_mother>,<name_of_child>,<sex>).
```

Define the following new predicates based on the **child** facts:

> **father(X,Y)**, meaning **X** is the father of **Y**
> **mother(X,Y)**, meaning **X** is the mother of **Y**
> **son(X,Y)**, meaning **X** is the son of **Y**
> **grandfather(X,Y)**, meaning **X** is the grandfather of **Y**
> **sister(X,Y)**, meaning **X** is the sister of **Y**
> **uncle(X,Y)**, meaning **X** is the uncle of **Y**
> **ancestor(X,Y)**, meaning **X** is the ancestor of **Y**

half_sister(X,Y), meaning **X** is the half-sister of **Y**

4-4. (P) Figure 4-9 is a picture of three stacks of blocks on a table.

(a) Represent in Prolog the facts describing the relationships between the blocks. Use two predicates, one that says a block is on another, and one that says a block is immediately to the left of another. Only give facts of the second type for blocks at the bottom of piles.

(b) Type your facts into a file.

(c) Now ask the Prolog interpreter:

--what blocks are on block B;
--what blocks block A is on;
--what blocks are on other blocks;
--what blocks are on blocks that are immediately to the left of other blocks.

(d) Define a new predicate **above** which is true when a block is anywhere in the stack above another block. Define a new predicate **stackleft** when a block is in a stack that is immediately to the left of the stack of another block. Put these into your file, and load the whole thing.

(e) Now ask the Prolog interpreter:

--what blocks are above other blocks;
--what blocks are either above block F or in a stack immediately to the left of its stack;
--what blocks are above other blocks but are not in a stack immediately to the left of any block.

4-5. (A) Consider the rules:

```
a(X) :- not(b(X)).
```

```
b(X) :- not(c(X)).
```

Assuming predicate **b** is never referenced except in the preceding, and is never queried directly, and there are no facts with predicate **b**, would it be equivalent to define a single rule

```
a(X) :- c(X).
```

Why?

4-6. (A) (a) Consider the following to be true:

> Clint is the mayor.
> The mayor is respected.

From this we can conclude that:

> Clint is respected.

But now suppose:

> Clint is the mayor.
> The mayor is an elected office.

From this we would seem to be able to conclude:

> Clint is an elected office.

What is the fallacy in reasoning?

(b) Consider the following to be true:

> Clint is a movie star.
> Clint is mayor.

From this it makes sense to conclude:

> A movie star is mayor.

But now suppose as true:

> John is a little stupid.
> John is mayor.

It is fallacious then to conclude:

> A little stupid is mayor.

What is the fallacy?

4-7. (A,E) Explain using Prolog concepts the different meanings of the word "and" in the following.

(a) A food crop is anything that is a plant and provides substantial human nutrition.

(b) A department chairman is a professor and is responsible to the provost.

(c) To put out a fire, your options are to apply flame retardants and to let it burn itself out.

(d) Tom and Sue are managers.

(e) Tom and Sue are friends.

4-8. Consider this query:

```
?- a(X,Y), b(X,Y).
```

used with this database:

```
a(1,2).
a(3,5).
a(R,S) :- b(R,S), b(S,R).


b(1,3).
b(2,3).
b(3,T) :- b(2,T), b(1,T).
```

Without using the computer, what is the first answer found to the query?

4-9. Inheritance involves a thing and a property. Suppose the thing is a computer program. Give two different examples of relationship predicates that are involved in inheritance, and the corresponding properties.

4-10. (A) Consider the type predicate **some** of two arguments **<set>** and **<property>** which is true whenever some members of **<set>** have **<property>**. So for instance **some(people,american)** means that some people are American. Consider inheritance involving the "set containment" relationship predicate.

(a) Does **some** inherit upward (to an including set), downward (to an included set), both, or neither? (Note: inheritance is only "yes" or "no", never "maybe".)

(b) Write the inheritance for part (a) as a Prolog rule.

(c) Consider the similar predicate **all** which is true whenever all members of **<set>** have **<property>**. So for instance **all(people,american)** means that all people are American. Does it inherit upward, downward, both, or neither?

(d) Consider the similar predicate **most** which is true whenever most members of **<set>** have **<property>**. Does it inherit upward, downward, both, or neither?

4-11. (R,A) Suppose we have facts about the components of a car. Suppose:

> --**front_of(X,Y)** means part **X** of the car is in front of part **Y** (towards the headlights);
> --**inside_of(X,Y)** means part **X** of the car is inside (contained in) part **Y**.

(a) Is **inside_of** transitive?

(b) Does **front_of** inherit with respect to **inside_of**? If so, in which direction?

(c) Why could it be more useful for a program to have a description of where things are under the hood of a car in terms of **front_of**, **right_of**, and **above** facts instead of in terms of Cartesian (x, y, and z) coordinates?

4-12. Consider the accounting department of some organization you know. For each of the following properties, say whether it inherits (1) upward, to some entity including the accounting department, (2) downward, to some entity included in the accounting department, (3) both directions, or (4) neither direction.

(a) There are crooks in it.

(b) Half the employees in it are crooks.

(c) Regulations controlling crookedness are enforced.

(d) All crooks in it have been caught and punished.

4-13. (R,A,P) (a) Represent the following facts in Prolog (binary-relationship predicates are recommended). Represent what the words mean, not what they say; each different word shouldn't necessarily be a different predicate name or argument. Type your Prolog facts into a computer file.

> An Acme hotplate has a cord and a body.
> Part of the body of an Acme hotplate is the heating element.
> The heating element is metal.
> Another part of the body of an Acme hotplate is the cover.

The cover has a knob.
Plastic is always used for knobs.
One of the things a cord consists of is a wire.
Metal comprises the wire.
Part of the cord is an insulater.
The insulater is fiber.

(b) Start up the Prolog interpreter, load the file, and ask

--what things are stated to contain metal;
--what things are stated to be parts of the body of a hot plate.

(c) Define rules for transitivity of **part_of** and upward inheritance of material with respect to **part_of**, rules that can be applied to your answer to part (a). Put these in the file.

(d) Now ask:

--what things contain plastic;
--what things do not contain fiber;
--what things containing metal also contain fiber.

4-14. Suppose we have **a_kind_of** facts about 31 objects. Suppose by transitivity on these facts we can show that object **a** is a kind of object **b**. Suppose the facts have no redundancy (two different routes between the same two objects by following arrows) and no cycles (ways to leave an object and return by following arrows).

(a) What is the minimum number of times that you could successfully use the recursive transitivity rule (the rule, not any fact) proving this? Use the two-rule form in Section 4.10.

(b) What is the maximum number?

(c) Suppose 30 of the 31 objects appear once and only once each as the first argument to the **a_kind_of** facts. Suppose 16 of the 31 objects never appear as a second argument and suppose the rest appear exactly twice each as a second argument. Of the remaining 15 that do appear as a second argument, 8 of them appear in facts with the 16; of the remaining 7, 4 appear in facts with the 8; of the remaining 3, 2 appear in facts with the 4; and the remaining one appears in facts with the 2. What data structure does the semantic network resemble?

(d) For the situation in part (c), what is the maximum number of times the recursive transitivity rule could be successfully used to prove that object **a** is a kind of object **b**?

(e) Suppose **b** has property **v**, and suppose that **a** inherits this value because **a** is a kind of **b**. Assume as before there are 31 objects total. What is the maximum number of times the inheritance rule (of the form of the first one in Section 4.9) could be successfully used proving this?

4-15. (a) Represent the following as Prolog facts and rules (definitions). Hint: represent what these mean, not what they literally say. Be as general as you can.

> A VW Rabbit is a VW.
> Tom's car is a VW Rabbit.
> Dick's car is a VW Rabbit.
> A VW has an electrical system.
> Part of the electrical system is the alternator.
> The alternator is defective on every VW.

(b) Write Prolog inference rules that will allow conclusion that Tom's car or Dick's car is defective. Hint: you need to define transitivity and inheritance for concepts in part (a).

(c) Prove that Dick's car is defective, given the facts and rules of parts (a) and (b). (Don't prove it the way a Prolog interpreter would--omit dead ends.)

4-16. (P) Write Prolog definitions for the California speed laws that follow, as extracted from the *California Driver's Handbook*. Your top-level predicate should be called **limit**, with one argument, an output variable. The program should set that variable to the legal maximum speed under the current conditions. The current conditions should be specified as Prolog facts.

Don't worry too much about the predicates you use here; there's much room for personal taste. Instead, worry about the *order* of Prolog definitions and facts. Note you must handle the situation in which you may have seen several road signs recently, some perhaps contradictory, and you must decide which ones apply. Assume though that any other facts (like the presence of children) apply to the immediate vicinity. Warning: you'll find the laws unclear about certain situations; just pick something reasonable in those cases.

> "The maximum speed limit in California is 55 miles per hour. Other speed limit signs tell you the highest speed at which you can expect to drive with safety in the places where the signs are set up...."

> "In business or residence districts, 25 miles per hour is the speed limit unless signs show other limits. When you see a "SCHOOL" sign, the speed limit is 25 miles per hour while children are outside or are crossing the street during school hours. The 25 m.p.h. limit applies at all times when a school ground is unfenced and children are outside, even though the road is posted for a higher speed. Lower speed must be obeyed if posted...."

> "When you come within 100 feet of a railroad crossing and you cannot see the tracks for

> 400 feet in both directions, the limit is 15 m.p.h. This limit does not apply if the crossing is controlled by gates, a warning signal or a flagman."
>
> "The 15 m.p.h. limit also applies at blind intersections where you cannot see for 100 feet both ways during the last 100 feet before crossing, unless yield or stop signs on the side streets give you the right of way--also in any alley."

As tests, run the program to find the limit at each point in the following scenario:

(a) You enter a residential district. A sign says 55 m.p.h.

(b) You are still in the residential district. You come to a SCHOOL sign, and students are on the sidewalks. The time is within school hours.

(c) A speed limit sign says 35 m.p.h. You enter an alley.

4-17. (E) Explain in what ways legal definitions are different from Prolog definitions. Is this a weakness of legal definitions, a weakness of Prolog, both, or neither? Should the two evolve closer together?

4-18. (E) Definitions of terms in politics don't seem to be very much like Prolog terms. For instance, what one nation calls an unprovoked military attack may be considered by the attacking nation as "claiming what is rightfully theirs" or "preventing terrorism". These excuses are not arbitrary but are supported by dictionary definitions. What makes political definitions so much more slippery than the Prolog definitions of this chapter?

4-19. There are several part-whole relationships, even for the same object. Consider a piece of rock consisting of | 10 sup 24 | molecules of silicon dioxide, whose chemical formula is |Si O sub 2|.

(a) Give an interpretation of **part_of** for which the **color** property inherits to any part of this rock.

(b) Give an interpretation of **part_of** for which the **number_of_molecules** property inherits to any subpart.

4-20. Consider the following proof that God does not exist. Take the following statement as true from the definition of God:

> God saves those who can't save themselves.

(a) Write this as a Prolog rule whose left side and right side both refer to a **saves** predicate of two arguments.

(b) Suppose the person saved is God. Show the bindings in the rule, and explain what the rule becomes in this case.

(c) As you can see, there's a serious logical problem here. Does this prove God doesn't exist? Why?

Go to book index

Arithmetic and lists in Prolog

Before going any further, we need to introduce two additional features of Prolog that will come in handy in writing future programs, arithmetic and lists. These give rules new capabilities. As we've seen already, rules can:

1. define new predicates in terms of existing predicates
2. extend the power of existing predicates (as with inheritance rules)
3. recommend what to do in a situation (as with the traffic lights program)

To these, we'll now add:

4. quantify and rank things
5. store, retrieve, and manipulate sets and sequences of data items

Arithmetic comparisons

Prolog has built-in arithmetic comparison predicates. But their predicate expressions are written differently from those shown so far: they're written in the *infix* notation of mathematics. The predicate name comes between the arguments, like this:

3 > **4** means 3 is greater than 4
15 = 15 means 15 equals 15
X < Y means **X** is less than **Y**
Z >= 4 means **Z** is greater than or equal to 4
PPPP =< 3 means PPPP is less than or equal to 3

We'll usually put spaces around infix symbols to make them easier to see, but it's not required. As an example, here's the definition of a predicate that checks if a number is positive:

```
positive(X) :- X > 0.
```

With this definition in our database, it could be used like this:

```
?- positive(3).
yes
?- positive(-6).
no
```

Here's the definition of a predicate that checks if its first argument is a number lying in the range from its second to its third argument, assuming all arguments are bound:

```
in_range(X,Y,Z) :- X >= Y, X =< Z.
```

Using this definition, the query

```
?- in_range(3,0,10).
```

gives the response **yes**.

Arithmetic assignment

Like any computer language, Prolog has arithmetic computations and assignment statements. Arithmetic assignment is done by expressions with the infix **is** predicate. Querying these peculiar expressions has the side effect of binding some variable to the result of some arithmetic computation. For instance

```
X is ( 2 * 3 ) + 7
```

binds (assigns) **X** to the value 13 (2 times 3 plus 7). The thing to the left of the **is** must be a variable name, and the stuff to the right must be an algebraic formula of variables and numeric constants, something that evaluates to a number. The algebraic formula is written in standard infix form, with operations **+**, (addition), **-** (subtraction), ***** (multiplication), and **/** (division). We'll often put spaces around these symbols to make them more readable. The algebraic formula can have variables only if they're bound to values, as in

```
Y is 2, X is Y * Y.
```

where **Y** is first bound to 2, and then **X** is bound to 4. A practical example is this definition of the square of a number, intended to be a function predicate:

```
square(X,Y) :- Y is X * X.
```

If this rule is in the Prolog database, then if we query

```
?- square(3,Y).
```

(that is, if we ask what the square of 3 is), the Prolog interpreter will type

```
Y=9
```

Notice that since predicate expressions aren't functions in Prolog, we can't write anything like

```
f(X,Y) + g(X,Z)
```

even if **f** and **g** are function predicates, because expressions only succeed or fail; expressions don't have "values". Instead, to add the two function values we must say something like

```
f(X,Y), g(X,Z), T is Y + Z
```

Another warning: don't confuse = with **is**. The = is a purely logical comparison of whether two things are equal. (Originally intended for numbers, it also works for words.) The **is** is an operation, an arithmetic assignment statement that figures out a value and binds a variable to it.

Reversing the "is"

A serious weakness of arithmetic, which makes it different from everything else in Prolog we've talked about so far, is that it isn't multiway or reversible. For instance, if we have the preceding definition of **square** in our database, and we query

```
?- square(X,9).
```

wondering what number squared is 9, the interpreter will refuse to do anything because the right side of the **is** statement refers to an unbound variable. This is different from having a bunch of arithmetic facts in *prefix* form like

```
square(0,1).
square(1,1).
square(2,4).
square(3,9).
```

for which we could query **square(3,Y)** or **square(X,9)** or even **square(X,Y)** and get an answer. Similarly, for the preceding definition of **positive**, the query

```
?- positive(X).
```

won't work: the interpreter can only do a > comparison when both things are bound to numbers. So it will complain and refuse to do anything.

The Prolog interpreter's excuse for its behavior is that function inversion and other such *multiway reasoning* is hard to do in general, and sometimes is impossible. A square of a number is easy to compute, but a square root requires iterative approximation and a lot more code. And there are an infinity of positive numbers; where should an interpreter start when asked to give one? Artificial intelligence requires flexible reasoning capable of going in many different directions--people seem to do it. So it's desirable to get around the interpreter's limitations.

One way is to provide additional rules for a predicate definition. Helpful in this is the built-in Prolog predicate **var** of one argument, which succeeds if that argument is an unbound variable, and fails otherwise. As an example of its use, consider a **better_add** predicate of three arguments which says the sum of the first two arguments is the third argument. If all three arguments are bound (inputs), then it checks the addition. If the first two arguments are bound, it binds the third to their sum. If the first and third arguments are bound, it binds the second to the difference of the third and the first. Similarly if the second and third arguments are bound, it binds the first to the difference of the third and second. Here's the code (**Z** is a temporary-storage variable):

```
better_add(X,Y,S) :- not(var(X)), not(var(Y)), not(var(S)),
  Z is X + Y, Z=S.
better_add(X,Y,S) :- not(var(X)), not(var(Y)), var(S),
  S is X + Y.
better_add(X,Y,S) :- not(var(X)), var(Y), not(var(S)),
  Y is S - X.
better_add(X,Y,S) :- var(X), not(var(Y)), not(var(S)),
  X is S - Y.
```

We can't handle two arguments unbound; then there's an infinite number of possibilities for the bindings. But at least the preceding handles three more cases than the Prolog **is** can handle by itself.

The **in_range** predicate of Section 5.1 can provide another example of a predicate enhancement. That predicate checked whether its first argument (an input number) was between the second and third arguments (input numbers too). We can improve **in_range** so that an unbound first argument will make it *generate* a number between other two arguments, and generate further numbers on backtracking. To make things easier, we'll assume all numbers will be integers. Here's the definition of this **integer_in_range**:

```
integer_in_range(X,Y,Z) :- not(var(X)), not(var(Y)), not(var(Z)),
  X >= Y, X =< Z.
integer_in_range(X,Y,Z) :- var(X), not(var(Y)), not(var(Z)),
  Y =< Z, X is Y.
integer_in_range(X,Y,Z) :- Y =< Z, Y2 is Y + 1,
integer_in_range(X,Y2,Z).
```

This is a *tail-recursive* program of a form we'll use many times in this chapter. (Again, see Appendix B to review recursion.) The first rule handles the case handled before. The second rule says if **X** is unbound and **Y** and **Z** are bound, and we want to generate an integer on the range **Y** to **Z**, we can always pick **Y**. Otherwise (if a semicolon is typed), the third rule is used. It "crosses out" **Y** from the range by increasing the lower limit of the range by 1, and generates an integer from this new, narrower range. If the range ever decreases so much that it disappears, all the rules fail. So if we query

```
?- integer_in_range(X,1,10).
```

the interpreter first replies **X=1**; then if we type a semicolon, **X=2**; then if we type a semicolon, **X=3**; and so on up to 10.

Lists in Prolog

Another important feature of Prolog is linked-lists. Every argument in a predicate expression in a query must be anticipated and planned for. To handle sets and sequences of varying or unknown length, we need something else: linked-lists, which we'll henceforth call just *lists*.

Lists have always been important in artificial intelligence. Lisp, the other major artificial intelligence programming language, is almost entirely implemented with lists--even programs are lists in Lisp. The extra space that lists need compared to arrays (see Appendix C) is more than compensated in artificial intelligence applications by the flexibility possible.

Square brackets indicate a Prolog list, with commas separating items. For example:

```
[monday,tuesday,wednesday,thursday,friday,saturday,sunday]
```

(Don't confuse square brackets "**[]**" with parentheses "**()**"; they're completely different in Prolog. Brackets group lists and parentheses group arguments.) Lists can be values of variables just like words and numbers. Suppose we have the following facts:

```
weekdays([monday,tuesday,wednesday,thursday,friday]).
weekends([saturday,sunday]).
```

Then to ask what days are weekdays, we type the query

```
?- weekdays(Days).
```

and the answer is

```
Days=[monday,tuesday,wednesday,thursday,friday]
```

We can also bind variables to items of lists. For instance, if we query

```
?- weekends([X,Y]).
```

with the preceding facts in the database, we get

```
X=saturday, Y=sunday
```

But that last query requires that the weekends list have exactly two items; if we query

```
?- weekends([X,Y,Z]).
```

we get **no** because the query list can't be made to match the data list by some binding.

We can work with lists of arbitrary length by the standard methods for linked-pointer list manipulation. We can refer to any list of one or more items as **[X|Y]**, where **X** is the first item and **Y** is the rest of the list (that is, the list of everything but the first item in the same order) | REFERENCE 1|. .FS | REFERENCE 1| In the language Lisp, **X** is called the *car* and **Y** is called *cdr* of the list. .FE We'll call "|" the *bar* symbol. Note that **[X|Y]** is quite different from **[X,Y]**; the first can have any nonzero number of items, whereas the second must have exactly two items. Note also that **X** and **Y** are different data types in **[X|Y]**; **X** is a single item, but **Y** is a list of items. So **[X|Y]** represents an uneven division of a list.

Here are some examples with the previous weekdays and weekends facts.

```
?- weekdays([A|L]).
A=monday, L=[tuesday,wednesday,thursday,friday]

?- weekdays([A,B,C|L]).
A=monday, B=tuesday, C=wednesday, L=[thursday,friday]

?- weekends([A,B|L]).
A=saturday, B=sunday, L=[]
```

The "**[]**" is the list of zero items, the *empty list* | REFERENCE 2|. .FS | REFERENCE 2| Called *nil* in the language Lisp. .FE

Defining some list-processing predicates

Let's write some famous list-processing programs (summarized in Figure 5-1). Programs requiring many lines of code in conventional programming languages can often be quite short in Prolog because of its declarative nature. We'll define mostly function predicates. Following the convention of Section 2.9, the function result is the last argument of their predicate expressions.

First, here's a definition of a predicate that computes the first item of an indefinitely long list:

```
first([X|L],X).
```

This definition is a fact, not a rule--but remember, facts are just rules with no right side. So **X** stands for any item, and **L** stands for any list.

Here's a definition of the last item of a list:

```
last([X],X).
last([X|L],X2) :- last(L,X2).
```

The first line says that the last item of a list of one item is that item. The second line says the last item of any other nonempty list is the last item of the list formed by removing the first item. This is a tail-recursive program with the first line the *basis step* (simple nonrecursive case) and the second line the *induction step* (recursive case). Tail recursion is the standard way to define list-processing predicates in Prolog, with each recursion chopping one item off a list.

We can use **first** and **last** just like the predicate definitions in Chapter 4, to work on data we type in. For instance:

```
?- last([monday,tuesday,wednesday,thursday,friday],X).
X=friday
```

We can also use it on lists in the database by doing a database retrieval first. Suppose we have a database fact:

```
weekdays([monday,tuesday,wednesday,thursday,friday]).
```

Then we could find out the last weekday by:

```
?- weekdays(L), last(L,X).
L=[monday,tuesday,wednesday,thursday,friday], X=friday
```

As another example of a list-processing definition, consider **member(X,L)** which is true if item **X** is a member of list **L**. We can give the following fact and rule (note the order: the recursive part of a definition should always come after the nonrecursive, to avoid an infinite loop) | REFERENCE 3|:

```
member(X,[X|L]).
member(X,[Y|L]) :- member(X,L).
```

.FS | REFERENCE 3| Recursive list-processing predicate definitions, and many other recursive definitions too, can be made considerably more efficient by Prolog's built-in "cut" predicate (symbolized by "**!**"), to be explained in Section 10.7. For instance, a better version of **member** for most purposes is: **member(X,[X|L]) :- !.**
member(X,[Y|L]) :- member(X,L). .FE The fact says that **X** is a member of any list where it is the first item; there's no need to check the rest of the list then. Otherwise, figure out if **X** is a member of the rest of the list, and that's the answer. Notice there's no need to give conditions under which **member** fails, like

```
member(X,[]) :- 1=2.
```

Since 1 can never equal 2, this rule never succeeds. But in Prolog, failure failing when we use a rule means the same thing as no rule at all. So the immediately preceding rule is completely useless.

We just gave a declarative explanation of the **member** predicate. For a *procedural* explanation (though we emphasize again that this is not the best way to understand recursive programs), consider the query

```
?- member(dick,[tom,dick,harry]).
```

The first line of the **member** definition fails because **dick** is not **tom**. So the second line is used, creating a recursive call

```
?- member(dick,[dick,harry]).
```

for which the first line succeeds. So the original query gives **yes**.

The **member** definition will also work when the first argument is unbound (an output). Then the program *generates* members of a list in succession under backtracking, something quite useful for artificial-intelligence programs. Consider the query:

```
?- member(X,[tom,dick,harry]).
```

When the interpreter executes this, the first line of the program can match **X=tom**, and this binding is printed out. If we now type a semicolon, we request a different binding, forcing the interpreter to use the second rule. So this recursive call is executed:

```
?- member(X,[dick,harry]).
```

And for this new query the first line can succeed, giving the result **X=dick**. If we type another semicolon, we'll be querying

```
?- member(X,[harry]).
```

and we'll get **X=harry**; and if we type yet another semicolon, we'll be querying

```
?- member(X,[]).
```

and we'll get **no**.

Here's a predicate **length(L,N)** that computes length **N** of a list **L**:

```
length([],0).
length([X|L],N) :- length(L,N2), N is N2 + 1.
```

Remember, **[]** represents the *empty list*, the list with no members. The first line says the empty list has length zero. The second line says that the length of any other list is just one more than the length of the list created by removing the first item. For instance:

```
?- length([a,b,c,d],N).
N=4
```

Here's a predicate **max(L,M)** that computes the maximum of a list **L** of numbers:

```
max([X],X).
max([X|L],X) :- max(L,M), X > M.
max([X|L],M) :- max(L,M), X =< M.
```

The first line says the maximum of a list of one item is that item. The second line says that the first number in a list is the maximum of the list if it's greater than the maximum for the rest of the list. The third line says the maximum of a list is the maximum for all but the first item of the list if neither of the first two rules applies. For instance:

```
?- max([3,7,2,6,1],N).
N=7
```

List-creating predicates

Suppose we want to delete every occurrence of some item from a list, creating a new list. We can do this with a predicate **delete** of three arguments: (1) the item **X** we want to get rid of (an input), (2) the initial list **L** (an input), and (3) the final list **M** (an output). And we'll assume that's the only pattern of inputs and outputs we'll ever use. For instance:

```
?- delete(b,[b,a,b,b,c],M).
M=[a,c]
```

To define this, we could write:

```
delete(X,[],[]).
delete(X,[X|L],M) :- delete(X,L,M).
delete(X,[Y|L],Mnew) :- not(X=Y), delete(X,L,M), Mnew is [Y|M].
```

But there's a better way to write the last rule: we can move the **[Y|M]** list to the left side. This is good because (1) the **is** is unnecessary because left sides can also bind variables, and (2) **is** isn't completely reversible, and we'd like a more multiway program. So we could use instead:

```
delete(X,[],[]).
delete(X,[X|L],M) :- delete(X,L,M).
delete(X,[Y|L],[Y|M]) :- not(X=Y), delete(X,L,M).
```

This works the same, even with the third argument unbound, because nothing can be done with the **[Y|M]** on the left side until the right side is executed and **M** is bound. So the construction of **[Y|M]** remains the last thing done by the third rule.

You may be puzzled why the **not(X=Y)** in the third line is necessary. We could write

```
delete(X,[],[]).
delete(X,[X|L],M) :- delete(X,L,M).
delete(X,[Y|L],[Y|M]) :- delete(X,L,M).
```

The **delete** predicate never fails for its first two arguments bound; one of those three rules must always succeed. So if the second line fails, the left side must be at fault, and **X** and **Y** must be different, right? Yes, but only the *first* time through. If we ever backtrack into this **delete**, we'll be in trouble because backtracking would use the third rule for some situation in which it used the second rule previously. For instance:

```
?- delete(b,[a,b,a,b,c],L).
L=[a,a,c] ;
L=[a,a,b,c] ;
L=[a,b,a,b,c] ;
no
```

So be careful in Prolog programming: just because something works OK for its first answer doesn't mean it will work OK on backtracking to get new answers.

Next, here's a useful predicate that "appends" (concatenates) one list to another. It has three arguments: the first list, the second list, and the combined list.

```
append([],L,L).
append([X|L1],L2,Lnew) :- append(L1,L2,L3), Lnew is [X|L3].
```

As with **delete**, we can rewrite the last rule to eliminate the awkward and undesirable **is**, moving the **[X|L3]** to the left side:

```
append([],L,L).
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).
```

This says first that the anything appended on the empty list is that thing itself. Otherwise, to append some nonempty list having first item **X** to a second list **L2**, append the *rest* of that first list (without **X**) to **L2**, and then put **X** in front of that. Study this revised definition carefully; it's a good example of how the style of Prolog programming differs from the style of most other programming.

Figure 5-2 shows an example using **append** with the first two arguments bound (inputs) and the third argument unbound (an output), for the query

```
?- append([gas,oil],[tires,battery,radiator],Things_to_check_before_trip).
```

The nested boxes represent the rule invocation environments created with each recursion. The outer one holds the parameter and local variables for the initial invocation of **append**. This invocation recursively calls itself, creating the middle box, with its own distinct variables. This invocation of **append** in turn recursively calls itself, resulting in the inner environment (box) with yet more distinct variables. The processing state at this point is depicted in the Figure. Now:

1. The first line of **append** says to bind its third argument to its value of **L**, **[tires,battery,radiator]**, and the invocation of the inner box succeeds.

2. Returning next to the middle box, **[tires,battery,radiator]** is the value of **L3**, and its **X** is **oil**, so **[X|L3]** is **[oil,tires,battery,radiator]**. So the invocation of the middle box succeeds.

3. Returning to the outer box, **[oil,tires,battery,radiator]** is the value of *its* **L3**, so its **[X|L3]** is **[gas,oil,tires,battery,radiator]**. The outer box succeeds.

4. So the original third argument **Things_to_check_before_trip** is bound to **[gas,oil,tires,battery,radiator]**.

Like the **member** predicate and many other predicates defined without arithmetic, **append** will work several ways. In fact, it will work seven ways (see Figure 5-3). For instance, it will handle the case in which the third argument is bound (an input) but the first and second arguments are unbound (outputs). Then the first two arguments are bound to binary partitions (breakings-in-half) of the third argument. So

```
?- append(L1,L2,[tom,dick,harry]).
```

gives the following if you keep typing a semicolon:

```
L1=[], L2=[tom,dick,harry];
L1=[tom], L2=[dick,harry];
L1=[tom,dick], L2=[harry];
L1=[tom,dick,harry], L2=[];
no.
```

The other rows in Figure 5-3 show other ways **append** can be used. Basically, we've got seven quite different programs in one. This comes about from the declarative interpretation of the definition: it describes conditions that hold when its third argument is the result of appending the first two arguments, not how to do it. Again, it describes "what" instead of "how".

Combining list predicates

List predicate definitions can refer to other list predicates. For instance, we can use **member** to define a **subset** predicate that determines whether all the members of some list **L1** are members of some list **L2**. Here it is, and we print **member** as well to refresh your memory.

```
subset([],L).
subset([X|L1],L2] :- member(X,L2), subset(L1,L2).
member(X,[X|L]).
member(X,[Y|L]) :- member(X,L).
```

We leave a blank line between the rule groups with the same predicate name, following a convention (also followed in the traffic lights program of Section 4.11). Here's an example use of the program:

```
?- subset([b,c],[a,c,e,d,b]).
yes
```

Here's a program for sorting lists of numbers into increasing order, *insertion sort* in particular:

```
sort([],[]).
sort([X|L1],L2) :- sort(L1,L3), insert_item(X,L3,L2).


insert_item(X,[],[X]).
insert_item(X,[Y|L],[X,Y|L]) :- X < Y.
insert_item(X,[Y|L1],[Y|L2]) :- X >= Y, insert_item(X,L1,L2).
```

The first argument to **sort** is an unsorted input list, and the second argument is the output sorted list. The first argument to **insert_item** is an input item, the second argument an input list, and the third argument is the result of inserting that item into that list, an output. For instance:

```
?- sort([3,2,7,4],L).
L=[2,3,4,7]
```

Redundancy in definitions

Basis conditions in a recursive definition are simple cases that don't require recursion. We always need at least one basis condition in a recursion, but we can have more than one. For instance, we could define length of a list this way:

```
length([],0).
length([X],1).
length([X,Y],2).
length([X,Y,Z|L],N) :- length(L,N2), N is N2 + 3.
```

instead of equivalently:

```
length([],0).
length([X|L],N) :- length(L,N2), N is N2+1.
```

But the extra lines can speed calculation when our lists are very short on the average: we're able to answer many queries without recursion. Here's a similar alternative to the **member** definition of Section 5.5:

```
member(X,[X|L]).
member(X,[Y,X|L]).
member(X,[Z,Y|L]) :- member(X,L).
```

Here the second line is redundant, since its case can be covered by a slightly modified third line. But if lists are often, or the things we're looking for are usually towards the front of the lists, the preceding definition may be faster than the original definition.

Such modifications are a kind of *caching*, a concept that occurs in many disguises in artificial intelligence, and which will reappear in a quite different form in Chapter 6. Caching means asserting unnecessary or redundant facts to improve efficiency. The idea is to waste a little space (the extra facts) in the hope of improving calculation speed. Caching doesn't always improve speed significantly, so to justify it you first need to do experiments, or do some mathematical analysis like that we'll do in Section 9.14 and 13.3.

An example: dejargonizing bureaucratese (*)

Natural language (human language) is a major subarea of artificial intelligence research. Lists and list processing routines are the obvious way to represent and use sentences and languages in Prolog. As an example, consider a critical technical problem facing the United States today: the translation of bureaucratic jargon into real English. Bureaucratic organizations typically use their own terminology to make their accomplishments look a little less pathetic than they really are. It would be useful to take a sentence of such bureaucratic jargon, expressed as a Prolog list, and convert it to understandable everyday English. Such a translation program might be used routinely on government documents.

For instance, "impact" is often misused as a verb, as in "The study will impact the department." When so used, it can be replaced by the simpler and more standard English word "affect". Similarly, "adversely impact" and "negatively impact" can be replaced by "hurt". "Transition" is also misused as a verb, as in "The project will transition to phase 3," and can be replaced by "change". "Evaluate options" and "consider options" can be changed to "study", and "under advisement" and "under consideration" to "being studied". You can probably recall many more examples. These substitutions usually but not always have the same meanings, so the sentence created by applying them should always be carefully double-checked.

It's easy to write a Prolog program for this, once English sentences are represented in Prolog list format. First we represent the substitution pairs as facts. For example:

```
substitution([adversely,impact],[hurt]).
substitution([negatively,impact],[hurt]).
substitution([impact],[affect]),
substitution([will,transition],[will,change]).
substitution([must,transition],[must,change]).
substitution([to,transition],[to,change]).
substitution([consider,options],[study]).
substitution([evaluate,options],[study]).
substitution([under,advisement],[being,studied].
substitution([under,consideration],[being,studied].
substitution([expedite],[do]).
substitution([expeditiously],[fast]).
substitution([will,secure],[will,close]).
substitution([must,secure],[must,close]).
substitution([prioritize],[rank]).
```

The first argument contains the original words, and the second argument the words to be substituted. Note that extra words in the first argument narrow the applicability of the substitution, but reduce the possibility of making mistakes.

Next we define a predicate that recurses through the sentence list, like **member** and **delete** defined in Sections 5.5 and 5.6 respectively:

```
dejargonize([],[]).
dejargonize(L,NL) :- substitution(S,NS), append(S,L2,L),
  append(NS,L2,L3), dejargonize(L3,NL).
dejargonize([X|L],[X|L2]) :- dejargonize(L,L2).
```

The first line sets the basis condition as the empty list, and the last line recurses through the list. The middle two lines do the work of substitution. They check through the substitution facts for one whose first argument matches the front of the list (using the **append** predicate according to the third line of Figure 5-3, for second argument unbound), and substitute (using **append** according to the second line of Figure 5-3, for the third argument unbound), and recurse on the new list. Here's an example:

```
?- dejargonize
([we,must,consider,options,to,transition,expeditiously],L).
L=[we,must,study,to,change,fast]
```

Despite the somewhat frivolous nature of this program, the idea of substituting into strings of words is important in much natural language work, and we'll look some more into it at the end of the next chapter.

Keywords:

infix
arithmetic assignment
is
prefix
multiway reasoning
recursion
tail recursion

basis of a recursion

induction step of a recursion

lists

Lisp

linked-pointer list representation

the bar symbol

the member predicate

the delete predicate

the append predicate

caching

Exercises

5-1. (A,P) Define a Prolog predicate max(X,Y,Z,M) that says that M is the maximum of the three input numbers X, Y, **and Z. Use only ">" to compare numbers.**

5-2. (R,A) A student familiar with Pascal was writing a compiler in Prolog. This required translating an error code number into an English description, so they wrote rules like this:

```
translate(Code,Meaning) :- Code=1, Meaning is integer_overflow.
translate(Code,Meaning) :- Code=2, Meaning is division_by_zero.
translate(Code,Meaning) :- Code=3, Meaning is unknown_identifier.
```

This is poor Prolog programming style. How can it be improved?

5-3. (P) Write a better_divide like better_add that handles similar cases for division. Have the program prevent division by zero.

5-4. To figure out a tax amount, you subtract the deductions from the gross and multiply by the tax rate (expressed as a decimal number less than 1). Using the better_add predicate defined in Section 5.3 and an analogous predicate better_multiply that you define yourself, write a single Prolog rule that can be used to answer all the following questions by a single query each:

> **--If my gross is 1,000, my deductions 270, and my tax rate 0.15, what is my tax?**
>
> **--If my tax was 170 at a tax rate of 0.17, with no deductions, what was my gross?**
>
> **--If my tax was 170 at a tax rate 0.17 and a gross of 1500, what amount of deductions did I take?**
>
> **--What tax rate would result in a tax of 80 on a gross of 1200 with 400 in deductions?**

If your Prolog dialect can handle decimal numbers, show your program works correctly for the

preceding questions.

5-5. (P) (This requires a Prolog dialect with floating-point numbers.) Define a predicate square(X,Y) that says that its second argument is the square of its first argument (the result of multiplying the first argument by itself). Using the built-in var predicate, have it handle each of the four cases of binding of its arguments:

> **--if both arguments are bound, it checks that the second argument is the square of the first;**
>
> **--if the first argument is bound and the second argument unbound, it computes the square of the first argument;**
>
> **--if the first argument is unbound and the second argument is bound, it computes an approximation of the first argument within 0.001 by bisection search or some other iterative method from numerical analysis;**
>
> **--if both arguments are unbound, it generates all possible pairs of positive integers and their squares starting with 1.**

5-6. (P) (a) Define a new predicate integer_better_add, like better_add but able to handle the case in which its first two arguments are unbound (outputs), finding all pairs of integers that sum to a bound (input) integer third argument.

(b) Use part (a) to write a program to generate three-by-three magic squares that have a given number as characteristic sum. (A *magic square* is a two-dimensional array of integers such that the sum of eight things--the three columns, the three rows, and the two diagonals--is the same.)

5-7. Consider this query:

```
a(X,Y), not(c(X)), d(X,Y).
```

Suppose our Prolog database contains this, in order:

```
d(X,Y) :- X > 1, Y > 1.


a(0,1).
a(0,2).
a(2,1).
a(M,N) :- b(P,Q), b(Q,P), M is P + 1, N is Q + 1.


c(0).
```

```
b(3,1).
b(2,1).
b(1,2).
```

(a) Without using the computer, what is the first answer found to the query by a Prolog interpreter?

(b) Without using the computer, what is the second answer found to the query (when you type a semicolon after the first answer)?

5-8. (R,A) Suppose we have Prolog facts about named events in terms of the following three predicates:

> start(<event>,<time>), an event started at a particular time
> end(<event>,<time>), an event ended at a particular time
> duration(<event>,<length>), an event lasted for a particular length of time

We may have one, two, or all three of these facts about some event, and we can't know in advance which we will have if we have one or two.

(a) Write Prolog rules to infer an end time for some event when there is no end(Event,Time) fact for it, and to infer a start time when there is no start(Event,Time) fact for it.

(b) Define a new Prolog predicate after(Event1,Event2) which is true when its first argument Event1 definitely happened after its second argument Event2.

(c) Define a new Prolog predicate during(Event1,Event2) which is true when its first argument Event1 definitely happened while its second argument Event2 was happening.

(d) Explain where in a Prolog database of facts these rules should go for things to work right.

5-9. (R,A) The transmission of a car contains gears that transmit power to the car's wheels. You can infer the speed and direction of the wheels, given facts about what gears are connected to what other gears and what is driving them.

Assume gears are labeled g1, g2, g3, and so on. Assume the number of teeth on each gear is specified by facts of the form

```
teeth(<gear_name>,<number_of_teeth>).
```

Assume all rigid connections between gears on the same rigid shaft are specified by facts of the form

```
same_shaft(<gear_name_1>,<gear_name_2>).
```

Assume that all meshed (teeth-to-teeth) connections between gears are specified by facts of the form

```
meshed(<gear_name_1>,<gear_name_2>).
```

(a) We want to reason about the rotational speed and direction of gears. Give a good format for such facts for each gear.

(b) Anytime two gears are rigidly attached to the same rigid shaft, their rotational speeds are the same. Write a Prolog rule that can infer the rotational speed of one such gear on a shaft from the known rotational speed of another such gear. Use the fact format from part (a).

(c) Anytime two gears are connected or "meshed" the product of the number of teeth and the rotational speed for each gear is the same, except that one gear rotates in the opposite direction from the other. Write a Prolog rule that can infer the rotational speed of a gear from the rotational speed of a gear meshed with it, assuming the number of teeth on both gears is known. Use the fact format from part (a).

(d) Suppose gear g1 has a rotational speed of 5000 rpm in a clockwise direction. Suppose it is on the same shaft as g2, g2 is meshed to g3, and g2 is meshed to g4. Suppose g1 has 100 teeth, g2 has 30 teeth, g3 has 60 teeth, and g4 has 90 teeth. Give a Prolog query that will figure out the rotational speed and direction of gear g4 from a database of these facts. Then show the steps that the Prolog interpreter will take to answer that query. Note: you must write the facts with the correct order of arguments in order for your inference rules to apply properly to them.

(e) Explain how infinite loops could happen when reasoning about gears, if you weren't careful in specifying the facts.

(f) Suppose for some arrangement of gears and specified gear speeds you find gear g8 has a speed of 1200 rpm by one chain of reasoning, and 800 rpm by another chain of reasoning. What does this mean for the gears?

5-10. (P) Define a predicate convert that does units conversion for length measurement; for instance, it converts a measurement in feet to meters. The convert predicate should take four arguments: a number (an input), the units for that number (an input), the units to which you want to convert (an input), and the result number (an output). Handle the following units: meters, decimeters, centimeters, millimeters, decameters, kilometers, inches, feet, yards, and miles. Hint: don't write a separate rule for every pair of possible units, but chain inferences.

5-11. Consider representing maps of highway routes in Prolog. Suppose you don't have much memory space, so representing squares of the map as a two-dimensional array is out of the question. Instead, you want to store just information about what towns are connected by which highways segments (assume segments have names and numbers, not unique, like "California 62"), and distances. Suppose highway segments are in different counties, states, and countries, and we want to remember which. Suppose we also store the different maximum speed limits for different states and countries. Assume each route is in a single county (you can create imaginary towns on county lines to ensure this). Assume highway

segments meet only at towns.

(a) Give formats for the fact predicates you need.

(b) Give a rule or rules for inferring the maximum speed limit on a road.

(c) Give a rule or rules for inferring a distance (not necessarily the shortest, that's hard) between two towns. Don't worry about getting into a infinite loop; assume the Prolog interpreter is smart enough not to visit the same town twice in a route (we'll explain how to do this in Chapter 10).

(d) Suppose there are R routes (connecting two towns each), C counties, S states, and K countries. How many facts do you save with the inference in part (b)? Make reasonable assumptions if necessary.

(e) Suppose there are R routes (connecting two towns each) and T towns. Approximately how many facts do you save with the inference in part (c)? Perhaps consider different arrangements of towns. Make reasonable assumptions if necessary.

5-12. (P) Define a completely multidirectional inference_distance of three arguments. Its first argument is a kind of the second argument. Its third argument is the number of linking a_kind_of facts that must be followed to get from the first argument to the second. By "completely multidirectional" we mean able to handle any pattern of bindings of the arguments. Assume there are only a_kind_of facts, no rules, and the facts don't have any cycles, and there is only one route between any two things. (The inference distance concept is important in psychology, because some psychologists believe that humans have semantic networks in their heads and that the speed of human reasoning is proportional to the inference distance.)

5-13. (E) Consider a Prolog definition written to implement a function, in other words written to be used when all arguments but the last are bound (inputs), which binds its last argument to some unique value. In abstract mathematical terms, what characteristics must the function have to be easily used multidirectionally, that is with last argument bound and other arguments unbound?

5-14. (A,P) (a) Using a single call to append and no other predicates, implement the member predicate. (Hint: the fifth row of Figure 5-3 is closest to what you need.)

(b) Using a single call to append and no other predicates, implement the last predicate.

(c) Using just two calls to append and no other predicates, implement a deleteone predicate that removes a single occurrence of some item from a list.

(d) Using just two calls to append and no other predicates, implement a before predicate of three arguments, that succeeds if its first two arguments are both members of its list third argument, and where the first argument item occurs before the second argument item.

5-15. (E) Figure 5-3 is missing a row for the case in which all variables are unbound. Explain why for

this case the predicate definition will not work properly.

5-16. Consider the delete predicate defined in the chapter:

```
delete(X,[],[]).
delete(X,[X|L],M) :- delete(X,L,M).
delete(X,[Y|L],[Y|M]) :- not(X=Y), delete(X,L,M).
```

(a) Suppose you rewrite it as

```
delete(X,[],[]).
delete(X,[X|L],M) :- delete(X,L,M).
delete(X,[Y|L],[Y|M]) :- delete(X,L,M).
```

What happens when you query this predicate with second and third arguments bound and first argument unbound?

(b) What happens if you change the second line to

```
delete(X,[X|L],L).
```

and query this predicate with first and second arguments bound and third argument unbound?

5-17. (A) (a) Suppose that Prolog predicate mystery is queried with its first argument a bound (input) list and its second argument unbound (an output). Describe what the predicate mystery does, in a sentence of 20 or less English words. (Hint: try it on sample lists.)

```
mystery([],[]).
mystery([X],[X]).
mystery([X,Y|L],[X,censored|M]) :- mystery(L,M).
```

(b) Most recursive list-processing definitions have only one basis condition. Why does the preceding definition need two?

5-18. In the language Logo and other languages with *turtle geometry* primitives, there are special commands to control a plotting pen. The pen has a position (measured in millimeters in a coordinate system) and a direction it is pointing in the plane of the paper (measured in degrees). Two built-in predicates manipulate the pen: forward(X) which moves the pen forward a distance X in the direction it is pointing, and right(X), which turns the pen around X degrees (without moving its location) so it points in a new direction. This program draws spirals, increasing the length it moves forward each step:

```
spiral(Side,Angle,Increment) :- forward(Side), right(Angle),
  Side2 is Side + Increment, spiral(Side2,Angle,Increment).
```

(a) Draw what the query

```
?- unknown(4).
```

causes to be drawn. Assume the pen is pointing north at the start. Indicate the lengths of lines in your drawing. Here is the definition:

```
unknown(1) :- right(90).
unknown(N) :- Nm1 is N-1, unknown(Nm1), N2 is 5-N,
  forward(N2), unknown(Nm1).
```

(b) Draw what the query

```
?- flake(3).
```

causes to be drawn. Again assume the pen is pointing north at the start. Here is the definition:

```
flake(1) :- forward(1).
flake(N) :- Nm1 is N-1, flake(Nm1), right(-60), flake(Nm1),
  right(120), flake(Nm1), right(-60), flake(Nm1).
```

Hint: this one is too complicated to figure out procedurally--reason about the declarative meaning of the rules.

5-19. (A) Consider use of the member predicate with both arguments bound, and with the second (list) argument N items long.

(a) How many times is the rule invoked when the answer to the query is no?

(b) Suppose the answer to the query is yes and suppose the item being searched for is equally likely to appear at any position in the list. How many times on the average will the rule be invoked now?

5-20. Consider our "dejargonizing" program. Each substitution of one set of words for another takes one line to define. Explain how to compress this set of substitutions so one line can handle many substitutions. (Hint: variables can be items in lists.)

5-21. (E) What is wrong with the following "proof" that all horses are the same color? "One horse is obviously the same color as itself. Suppose for some N, every set of N horses is the same color. Then consider some horse Snowy, a white horse, for purposes of argument. Consider the set of N+1 items formed by including Snowy in an arbitrary set of N horses. Now if we take out some other horse than Snowy from this set (call him Blacky), we have a set of N horses. By the induction assumption, these horses are all the same color. But Snowy is white. Therefore all the other horses in the original set must have been white. So put Blacky back in and take out some other horse (besides Snowy, call him

Alabaster) to create a new set of N horses. This new set must also be all of the same color, so since it includes Snowy, the color must be white. But Blacky is in the set too, and so must be white too. Hence every set of N+1 horses must be white. Hence if any set of N horses is all the same color, any set of N+1 horses is of the same color. Hence by recursive argument, all horses are the same color."

5-22. (A,P) Write a modified transitivity rule (for let's say transitivity of a(X,Y)) that can't ever get into infinite loops when answering a query of the form

```
?- a(r,s).
```

Show your program working on some sample data that contains cycles with respect to the a predicate. (A "cycle" means a closed loop in the semantic network.) Hint: use an extra list argument.

5-23. (P) Create a Prolog database and query it appropriately to create poems of the following form:

> **A A B C C B D D D D E E B**

Here each capital letter stands for a class of nonsense words that must rhyme together. B represents one-syllable words, and the other letters represent two-syllable words. In addition, the poem cannot have duplicate words. Here is an example poem:

> **uga buga ru batta hatta nu fitty pitty witty ditty garra farra tu**

It will help to define a predicate different of two arguments that says whether two words are identical. Present your resulting poem as the value of a single variable, a list of lists for which each list is a line of the poem. Type semicolons to see what similar poems you get.

Go to book index

Control structures for rule-based systems

Rule-based systems for artificial intelligence (also called *production systems*, but that sounds like an assembly line) can work quite differently from the usual (built-in) way that Prolog interpreters work. We'll now discuss some of these ways. The differences are in the *control structure* or *conflict resolution* or *inference engine*, the way and order in which facts, rules, and parts of rules are used. Choices in the order for these things can enormously affect the success and efficiency of a rule-based system--Prolog interpreters aren't always best.

Control structures are somewhat like the algorithms in other parts of computer science. Control structures, however, are more general and less precise in their effects. As we said in Section 4.13, artificial intelligence programs tend to consist of many small pieces of code, and the control structure usually serves as a coordinator or manager for all these pieces instead of trying to impose a sequence on them all.

The control structure is critical for an important and practical class of rule-based systems. Called *rule-based expert systems*, these rule-based systems try to mimic the performance of human experts on specialized tasks. Their rules are usually more like recommendations than definitions, often like the traffic lights example of Section 4.11. Rule-based expert systems typically need large numbers of rules about a problem area or *domain* in order to approach skilled human performance. Because of their size, efficiency is important, and choice of control structure strongly affects efficiency. We'll frequently have expert systems in mind in this and the next two chapters, if not explicitly.

Just like everything else in computer science, there's a tradeoff of generality and flexibility against speed with control structure ideas. The ideas we'll discuss in this chapter can be placed on an flexibility/efficiency spectrum (see Figure 6-1). Those that are flexible, "high-level", and slow appear at the top, while those that are inflexible, "low-level", and fast (*compiled control structures*) appear at the bottom.

Backward-chaining control structures

Many control structures impose a single sequential ordering on everything that happens. But sequential control structures differ in the relative importance they assign to rule order, fact order, and query (*goal*) order. The usual Prolog-interpreter control structure described in Chapter 4 puts goal (query) order top priority, and then treats rule and fact order of equal importance after that, using the database order as a way of assigning priorities to individual rules and facts. This is *backward chaining* or *goal-directed reasoning*. It is told something to prove, and it tries to find a way, binding variables as necessary. If alternative conclusions are possible (as in the traffic lights example in which several actions are considered until one is found), backwards chaining can try to prove the first, then try the second if the first fails, and so on like an "or". Backward chaining is often a good control structure when there are many more facts than final conclusions (goals).

As an example, consider these rules, labeled with codes for easy reference:

```
/* R1 */ goal1 :- fact1.
/* R2 */ goal1 :- a, b.
/* R3 */ goal2(X) :- c(X).
/* R4 */ a :- not(d).
/* R5 */ b :- d.
/* R6 */ b :- e.
/* R7 */ c(2) :- not(e).
/* R8 */ d :- fact2, fact3.
/* R9 */ e :- fact2, fact4.
```

Suppose the goals in this rule-based system are **goal1** and **goal2(X)**, and suppose we consider them in that order. In other words, it's like a query was:

```
?- goal1; goal2(Z).
```

Suppose further that only facts **fact2** and **fact3** are true. R1 is tried first, but **fact1** isn't true so it fails. Next R2 is tried, invoking R4 and then R8. R8 succeeds because **fact2** and **fact3** are both true. So the subgoal **d** succeeds, and therefore R4 (and **a**) fails because of the meaning of **not**. So finally **goal2** is tried, which invokes R3. This invokes R7, then R9. The latter fails because **fact4** isn't true. So R7 succeeds, hence R3 succeeds with **X**=2, and hence **goal2(2)** succeeds.

There are several useful enhancements of backward chaining. A simple trick that can often greatly improve efficiency is to *cache* or enter as facts some or all of the conclusions reached. For instance, once we prove conclusion **b** with the preceding rules, we could add a **b** fact to the database. We should put fact **b** in front of rules that can prove it, so a subsequent query will find it before the rules. The more times **b** is queried later, the more effort this caching can save. The disadvantage is more facts to search through to answer questions.

Conclusion caching is simple with Prolog: just use the **asserta** built-in (that is, you don't have to define it) predicate of one argument, a fact to be asserted. This predicate **asserta** is like **not** in that the argument must be a predicate expression; querying it always succeeds, but has the side effect of adding that expression as a new fact to the database, in front of the facts with that same predicate name and rules with that name on their left side. Like most of Prolog's built-in predicates, **asserta** always fails on backtracking: there's no second way you can assert a fact. Prolog also has two related built-in predicates: **assertz** which caches a fact *after* the last fact or rule in the database having the same predicate name, and **retract** which removes (or "un-caches") a fact from the database.

Another trick to improve efficiency of backward chaining is not to require a complete starting database, but ask for facts as needed--that is, designate *virtual facts*. For instance, if there's something wrong with a car's electrical system, you could check for loose wiring. But that's a lot of work, and some wires are hard to find. We shouldn't require checking wires before running a car program, only if more obvious problems like a dead battery have been ruled out. Backward chaining works well with virtual facts because it only queries facts immediately relevant to its conclusions. The control structure we discuss next, forward chaining, does not have this advantage, but it has some others.

Forward chaining

Often rule-based systems work from just a few facts but are capable of reaching many possible conclusions. Examples are "sensory" and "diagnosis" expert systems, like those that identify an object from a description of what you see at a distance, or those that tell you what to do when your car breaks down from a description of what isn't working. For these, it often makes more sense to start with the facts and reason to goals (conclusions), what is known as *forward chaining* or *data-directed computation* or *modus ponens* reasoning.

As an example, take the rule

```
a :- b, c.
```

and suppose **b** and **c** are facts. Then we can conclude **a** is a fact, and add it to the facts we have. This is called *modus ponens* inference in logic. There's no query, no goals; we just use the facts we know to infer some new fact. (The built-in predicate **asserta** can help implement this too; see the next chapter.)

To use modus ponens as the basis of a control structure, take the facts in order. For each fact, find all rules whose right sides contain a predicate expression that can be matched to it. (We can index predicate names mentioned on right sides to speed this up.) Now "cross out" the predicate expressions matched in the rules; that is, create new rules like the old rules except without these expressions. But wherever a fact matched the last expression on the right side of some rule, the left side of the rule has been proved a new fact, after substituting in any bindings made on the right side; so cache that new fact in the database. The last paragraph gave an example. For another, consider:

```
a(X,3) :- b(X).
```

Then if **b(20)** is a fact, modus ponens would conclude **a(20,3)** is a fact.

It matters where we put a new fact among the others, since the facts are pursued in order. Usually it's good to put the new fact in front of those not yet considered, a *most-recent-fact* or *focus-of-attention* idea. So the fact just proved will be the next fact examined; the fact list then is much like a stack (last thing "in" is the first thing "out"). This might be good if we want to reach some particular conclusion as fast as possible. But, on the other hand, if we want to systematically find every conclusion possible from the facts, we should put new conclusions at the end of the set of facts; the fact list is then like a queue (last thing "in" is the last thing "out").

Any **not**s in rules require special handling. Since we want to follow the closed-world assumption for forward chaining too (it's simplest), we want **not** to mean "it can't be proved". So forward chaining must first assume all **not**s to be false, prove all possible facts, and only then consider as true those **not**s whose arguments are not now facts. Those **not**s may then prove new facts with new consequences. (To avoid such awkwardness, some artificial-intelligence systems let you state certain facts to be false, and we'll discuss how to handle such "unfacts" in Chapter 14, but this creates its own complications.)

Here's the formal algorithm for (pure) forward chaining:

1. Mark all facts as unused.

2. Until no more unused facts remain, pick the first-listed one; call it F. "Pursue" it:

   (a) For each rule R that can match F with a predicate expression on its right side, ignoring **not**s, and for each such match in the rule (there can be multiple match locations when variables are involved):

      (i) Create a new rule just like R except with the expression matching F removed. If variables had to be bound to make the match, substitute these bindings for the bound variables in the rule.

      (ii) If you've now removed the entire right side of rule R, you've proved a fact: the current left side. Enter that left side into the list of facts, and mark it "unused". (The focus-of-attention approach here puts the new fact in front of other unused facts.) Eliminate from further consideration all rules whose left sides are equivalent to (not just matchable to) the fact proved.

      (iii) Otherwise, if there's still some right side remaining, put the new simplified rule in front of the old rule. Cross out the old rule if it is now redundant. It is redundant if the old rule always succeeds whenever the new rule succeeds, which is true when no variables were bound to make the match.

   (b) Mark F as "used".

3. Create a new fact list consisting of every **not** expression mentioned in rules whose

argument does not match any used fact. Mark all these as "unused", and redo step 2.

A forward chaining example

Let's take an example of forward chaining with focus-of-attention handling of new facts. Consider the same rules used for backward chaining:

```
/* R1 */ goal1 :- fact1.
/* R2 */ goal1 :- a, b.
/* R3 */ goal2(X) :- c(X).
/* R4 */ a :- not(d).
/* R5 */ b :- d.
/* R6 */ b :- e.
/* R7 */ c(2) :- not(e).
/* R8 */ d :- fact2, fact3.
/* R9 */ e :- fact2, fact4.
```

Suppose the rules are taken in the given order; and that as before, only **fact2** and **fact3** are true, in that order (see Figure 6-2).

1. We start with **fact2**, and find the matching predicate expressions R8 and R9. This gives the new rules

```
/* R10 */ d :- fact3.
/* R11 */ e :- fact4.
```

Rules R8 and R9 are now redundant since no variables were bound, and R8 and R9 can be eliminated.

2. No new facts were discovered, so we pursue next **fact3**. This matches an expression in R10. So now R10 succeeds, and the new fact **d** is put in front of any remaining unused facts (though there aren't any now). R10 can be eliminated.

3. We pursue fact **d**, and find that rule R5 mentions it in its right side. (R4 mentions it too, but as **not(d)**, and we're saving **not**s for last.) Matching R5 gives the new fact **b**. Rules R5 and R6 can now be eliminated.

4. Fact **b** matches in R2, giving

```
/* R12 */ goal1 :- a.
```

and rule R2 can be eliminated. The current set of rules is:

```
/* R1 */ goal1 :- fact1.
/* R3 */ goal2(X) :- c(X).
/* R4 */ a :- not(d).
/* R7 */ c(2) :- not(e).
/* R11 */ e :- fact4.
/* R12 */ goal1 :- a.
```

5. We have no more facts to pursue. But we're not done yet, since R4 and R7 have **not**s.

6. Fact **d** is true, so rule R4 can't ever succeed. But fact **e** has not been proved. Hence add **not(e)** to the list of facts.

7. This matches the right side of R7. Hence **c(2)** is a fact too. Eliminate R7.

8. This matches the only expression on the right side of R3, when **X=2**, and hence **goal2(2)** is a fact. We can't eliminate R3 now because we had to bind a variable to make the match, and we can still use R3 for other values of **X**.

9. That's everything we can conclude.

Though Prolog interpreters don't *automatically* forward chain, it's not hard to teach them--see Section 7.10.

This "pure" forward chaining is rarer in applications than backward chaining. There's a story about a huge metal robot that came clanking into a bar one day. "I'm a pure-forward-chaining robot, and I can do a complete analysis of the quality of any liquor-dispensing establishment with a single sample. Please mix a martini for me, and pour it down the analysis chute in my chest." The bartender did so, and said, "That'll be eleven dollars. Say, we don't get too many forward-chaining robots in here." "At your prices I'm not surprised," replied the robot.

Hybrid control structures

Different control structure ideas can be combined in *hybrid* control structures. Hybrids of forward and backward chaining, compromising on the advantages and disadvantages of both, are often used, The most common is the *rule-cycle hybrid* | REFERENCE 1| because it is easy to implement (see Section 7.9). .FS | REFERENCE 1| The rule-cycle hybrid is often confused with pure forward chaining. .FE

With the rule-cycle hybrid, rules are tried in order as with backward chaining, but each rule is used in a forward chaining (modus ponens) way to assert new facts. The rule list is cycled through repeatedly, first ignoring any rules with **not**s. If the conditions on the right side of some rule all match facts (that's *facts*, not just something provable), then the rule succeeds and its left side (with appropriate bindings) is added to the database as a new fact. When no new rules succeed on a cycle through all of them, rules with **not**s are now considered; cycling resumes at the top of the rules, with the **not** expressions now succeeding if their arguments aren't now facts. Again we continue until no new rules succeed on a cycle. So with the rule-cycle hybrid, rule order takes precedence over fact order, but it's different than with backward chaining. Figure 6-3 summarizes the differences between forward chaining, backward chaining, and the rule-cycle hybrid.

Here's a more formal algorithm. Warning: it will only work with the restriction that no **not(p)** occurs in the right side of a rule before a rule having **p** as its left side, for any **p**, but that's usually easy to satisfy.

> Cycle through the rules repeatedly until no new facts are found on a cycle, ignoring rules with **not**s.
>
>> For each cycle, consider the rules in order.
>>
>>> For each rule R, treat its right side as a query about the facts (without using any other rules via backward chaining). If R succeeds, add its left side (with substitution of bindings made) as a fact at the front of the list of facts. And then eliminate from further consideration all rules whose left sides are equivalent to this new fact.
>
> Now repeat the previous step with *all* the original rules, taking also as true the **not**s whose arguments are not facts.

Take our standard example:

```
/* R1 */ goal1 :- fact1.
/* R2 */ goal1 :- a, b.
/* R3 */ goal2(X) :- c(X).
/* R4 */ a :- not(d).
/* R5 */ b :- d.
/* R6 */ b :- e.
/* R7 */ c(2) :- not(e).
/* R8 */ d :- fact2, fact3.
/* R9 */ e :- fact2, fact4.
```

With the rule-cycle hybrid when **fact2** and **fact3** are true (see Figure 6-4):

> 1. R1, R2, R3, R5 and R6 are tried (we skip R4 and R7 because they have **not**s). None succeed because nothing on any right side matches a fact that is stated to be true.

2. R8 is tried, and it succeeds. Fact **d** is asserted. Eliminate R8.

3. R9 fails.

4. We return to the top of the rule list and start a new cycle. Rules R1 through R3 fail as before, and R4 is ignored.

5. But R5 now succeeds since **d** is a fact. Fact **b** is asserted. Eliminate R5 and R6.

Now the rules are:

```
/* R1 */ goal1 :- fact1.
/* R2 */ goal1 :- a, b.
/* R3 */ goal2(X) :- c(X).
/* R4 */ a :- not(d).
/* R7 */ c(2) :- not(e).
/* R9 */ e :- fact2, fact4.
```

6. R7 and R9 fail (R8 was eliminated). And all the remaining rules fail on the next cycle.

7. Possibilities are exhausted, so we must now include rules with **not**s. R1, R2, and R3 fail as before, and R4 fails because **d** is a fact.

8. R5 and R6 were eliminated, and the **not** in R7 succeeds because **e** is not a fact. So **c(2)** is a fact. Eliminate R7.

9. None of R8, R9, R1, and R2 succeed. But R3 succeeds, with **X**=2, and **goal2(2)** must be a fact. We can't eliminate R3 because **goal2(2)** is more specific than the left side of R3. But we're done now if we only want to reach one goal.

A different hybrid of forward and backward chaining alternates (*time-shares*) between forward and backward chaining steps. It picks a fact, and finds what rule right sides mention it; it does backward chaining in those right sides to try to establish the left side as a fact. Then it picks another fact and does the same thing over again. This hybrid sometimes pays off when neither forward, backward, nor rule-cycle hybrid chaining works well.

Order variants

Query, fact, and rule ordering is important in backward, forward, and rule-cycle hybrid chaining, and control structures can apply many criteria to do it.

With backward chaining, predicate expressions in the query can be sorted by priority. In an "and", the expressions easiest to process or least likely to succeed can be done first. Sections 13.2 through 13.7 discuss these ideas at length.

With forward chaining, facts can be sorted by priority. The facts most useful (matchable in the right sides of the most rules) can go first, to help reach interesting conclusions faster. Or the facts most powerful in reaching conclusions can go first (statistics on queries can be kept for this).

Rule order is important to both forward and backward chaining. One common ordering is by *specificity*. Specificity doesn't mean how "concrete" the rules are, but whether conditions on the right side of rule 1 are a subset of conditions on the right side of rule 2; or in other words, whether the success of rule 2 means success of rule 1. The specificity ordering puts rules for the most narrowly described situations (or *exceptions*) first, then those for less narrow situations, then those still less narrow, and so on up to very broad *default* rules. (If the most broadest rules came first, the others would never get a chance to work.) Since catching subset relationships between rule right sides requires some analysis, a variant is to put the *longest* rules first, but this doesn't work quite as well.

Here's an example for rule-cycle-hybrid chaining in which all the rule left sides are goals:

```
/* R1 */ u :- b.
/* R2 */ v :- c.
/* R3 */ w :- b, c, d.
/* R4 */ x :- d, e.
/* R5 */ y :- b.
/* R6 */ z :- b, d.
```

A rule-specificity ordering would insist that R3 be in front of R1, R2, R5, and R6, and insist that R6 be in front of R1 and R5.

There are two problems with specificity ordering. The first is that there are often few such subset relationships among rules, leaving undecided how to complete the ordering. The second is that the narrowest rules apply rarely, meaning wasted work if they're first. Chapter 13 will study these issues. But usually a quick fix is available in the adding of extra **not** expressions to the broader rules, and putting those rules first. For instance, for our last example it may be possible to rewrite R1 as

```
/* R1 */ u :- b, not(c), not(d).
```

and then it doesn't matter where it goes.

This is often what the original rules really meant, since we often forget the minor exceptions to rules--there are just too many. For example, with the rule "if a car won't start and the radio won't play and the lights don't light, then the battery is dead", we don't rule out the chance that aliens from outer space have nullified all electric fields.

Partitioned control structures

When there are a thousand or more rules as in major current expert systems, the rules can interrelate in many ways, and a bug can be hard to trace. So just like large computer programs, it's a good idea to divide rules into groups, modules, or partitions for which members of each group have minimal interactions with members of other groups. You can think of each group as a separate rule-based system that may occasionally decide to call on another for a separate analysis. An advantage is that the rule groups can be written and debugged mostly separately. This idea is called a *partitioning* or *context-limiting* control structure.

Diagnosis expert systems provide good examples. For instance in diagnosis of a malfunctioning car, there are major systems of the car that don't interact much with one another. If electrical devices aren't working, you can be pretty sure that the problem is in the electrical system; or if the car goes forward but not backward, you can be pretty sure the problem is in the transmission. So you can put rules for electrical problems in one partition, and rules for transmission problems in another, rules for the engine and fuel system in another, rules for the car body in another, and so on. You'll need one other partition of rules, a "startup" partition, to look at key evidence and decide which partition appears most relevant to a problem. And partitions can choose to transfer control to another partition, if say none of their own rules succeed.

Meta-rules

A general approach can encompass all the control structure ideas so far: specification of control by a rule-based system itself. *Meta-rules* are just rules whose domain of knowledge is the operation of another rule-based system; they're a kind of *heuristic*, a topic we'll investigate more thoroughly in Chapter 9. Rules deciding to load partitions (Section 6.6) are one simple example of meta-rules, but they can do many other things. Remember that one goal for putting knowledge into computers was to make explicit the complicated "common-sense" knowledge people have but don't realize they have. How to order rules and use them is another kind of common-sense knowledge, also formalizable. Here are some example meta-rules for a backward-chaining-like rule-based system, to control selection of the next rule to try to satisfy instead of following the database order of rules:

> --Prefer the rule that handles the most serious issue.
> --Prefer the rule that was written by the most knowledgeable human.
> --Prefer the rule that is fastest to execute.
> --Prefer the rule that has been used successfully the most times.
> --Prefer the rule with the most things in common with the last rule successfully applied.

The big advantage of meta-rules is their flexibility and modifiability, which allows precise control of a rule-based system.

Meta-rules can express things besides rule orderings and partition referrals. Prolog interpreters make the *closed-world assumption* or *lack-of-knowledge inference*: if you can't prove something true, assume it false. This may be unfair for some predicates; a meta-rule could then override normal reasoning. For instance, a meta-rule could say to use the closed-world assumption only when querying predicates from a certain list, and to assume a failure means **yes** otherwise.

Meta-rules seem to be important in human reasoning. People aren't generally systematic enough to use any of the chaining methods successfully, but instead they rely on problem-specific meta-rules for deciding what to do next. So to reason more naturally, meta-rules are critical. But figuring out just what meta-rules people do use is hard.

Decision lattices

We'll now consider some lower-level control structure ideas: decision lattices, concurrency, and and-or-not lattices. In the terminology of computer science, these are *compiled* structures. But they're compiled in a different sense than what programming-language "compilers" produce: they represent a simplifying first step before the traditional compiler operates. Some people don't consider these compiled structures truly artificial intelligence, but they're so closely linked to artificial intelligence that we'd better explain them.

Choosing a good sequence for rules can be important and hard, as we discussed in Section 6.5. But computers can use storage structures besides sequences (see Appendix C). They can organize rules in a hierarchy, what is called a *decision lattice* or *discrimination net*. Decision lattices do a restricted but very efficient kind of reasoning, a kind of classification. The idea is to always specify where to go next in the computer based on question answers. In other words, a kind of *finite-state machine*. (Sometimes they're called *decision trees*, but technically they're lattices since branches that diverge can converge or "grow back together" later. Any graph without cycles in which this convergence can happen is a lattice and not a tree; cycles wouldn't make much sense here because you'd be asking the same question twice.)

For instance, consider an expert system to diagnose malfunctions of small household appliances (see Figure 6-5). It is important first to distinguish problems within the appliance from problems outside the appliance. A good way is to ask if the appliance works at all. If it doesn't, ask if it is plugged in. If it isn't, that is the problem. If it is, ask if other electric devices nearby (lights, clocks, etc.) work. If they don't, the problem sounds like a blown fuse. If other appliances definitely work, the problem must be internal to the faulty appliance. If no such observations can be made (as when there are no electrical appliances nearby), try plugging the faulty appliance into another outlet to see if the problem reappears.

On the other hand, if the appliance partially works, then it matters what kind of appliance it is. That's because interpretation of partial-failure clues is quite appliance-dependent, like smoke when the device has a heating element. As another example, strange noises are more serious in a device with no moving parts than in a blender. So the next question for a partially-working appliance should classify it.

So decision lattices impose a classification hierarchy on the universe based on observations. They are useful for simple expert systems, with several advantages:

1. Implementation is easy: just use pointers (memory references). They can even be implemented without a computer, as printed text with cross-references.

2. They need not explicitly question a human being: they can examine buffer contents or sensor readings. Then they can be fast, faster than the chaining methods, because no matching, binding, or backtracking is needed.

3. They can be designed to ask the absolutely minimal number of questions necessary to establish conclusions, unlike chaining methods for which such optimization can be difficult.

But decision lattices have major disadvantages as a compiled or "low-level" control structure:

1. They can't reason properly or efficiently for many applications because they don't easily permit variables or backtracking.

2. They are difficult to modify and debug, since later questions must assume certain results to earlier questions.

3. They can't easily reuse query answers since they don't explicitly cache.

4. They may be hard to build, because at each point you try to determine the best question to ask, something not so easy to judge.

Decision lattices were around long before computers. Expert-system technology only made significant progress when decision-lattice control structures were mostly abandoned, due to the limitations mentioned.

Concurrency in control structures

If speed of a rule-based system is important (as in a real-time application), and multiple processors are available, a control structure can use concurrency. Usually the processors must share and access the same database of facts and rules for this to work well. For a Prolog-style rule-based system, four types of parallelism for concurrency are identified (see Figure 6-6): (1) *partition parallelism*, (2) *or parallelism*, (3) *and parallelism*, and (4) *variable-matching parallelism*. These parallelisms are useful with all three kinds of chaining.

Partition parallelism means running different partitions of the rules simultaneously. Each partition can reason separately, though they can explicitly pass conclusions to one another, or cache into a global database. This is good if we've got groups of rules that don't interact much, each group relevant to a problem.

"And" parallelism is parallelism among expressions "and"ed on the right side of a rule or a query. Usually it is only done for the predicate expressions that do not bind variables, the "tests" in the generate-and-test concept (see Section 3.12). These tests can be done concurrently on separate processors; if any test fails, the whole "and" should fail, and the other processors should be sent a message to stop work. Otherwise, the "and" should succeed. "And" parallelism is probably not a good idea when some tests are much harder to satisfy than others; then the hard tests should go first (see Chapter 13).

"Or" parallelism usually means parallelism between rules with the same left-side predicate name. It is good when there are many such rule groups. Or-parallel rules are sometimes called *demons* because they're like little people each independently waiting for a particular set of conditions to be satisfied. "Or" parallelism can also mean parallel pursuit of facts in forward chaining.

Variable-matching parallelism is parallelism in the argument matching done when matching two predicate expressions to one another. It makes each match attempt faster, but it doesn't change the usual sequential examining of the database. It only pays off when you have a significant number of predicates with two or more arguments.

Concurrency can be simulated on a sequential machine. This gives a new class of control structures, the sequential reductions of concurrent process descriptions. This idea is often associated with the "agenda" search methods in Chapter 10.

Parallelism is not just an efficiency trick, however. Parallelism is necessary to model many real-world phenomena. These phenomena are often addressed in *object-oriented programming*, for which the world is divided into clusters of facts representing *objects*, each with its own partitioned module of rules governing its behavior. Object-oriented programming is especially useful for simulations. For instance, objects (and their facts) can represent organisms in an ecosystem, and rule modules for each kind of organism can govern the behavior of each object. Another application is to modeling components of a car, where each object represents a part of a car. We'll talk more about object-oriented programming in Chapter 12. While it emphasizes rule-partition parallelism, it can also involve the other three kinds. For instance in modeling organisms in an ecosystem, "and" and "or" parallelism can reflect the ability of organisms to do and think several things simultaneously.

And-or-not lattices

The extreme case of parallelism in rule-based systems is the *and-or-not lattice* representation of rules, in which each rule can be thought (or maybe actually is) a hardware logic gate incessantly computing a certain logical combination of input logic signals representing facts and intermediate conclusions. (It's sometimes incorrectly called an *and-or tree*, but like in the decision lattice, the paths can recombine after splitting, so it isn't a tree necessarily.) "And"s in a rule become "and" gates, "or"s become "or" gates, and "not"s becomes inverter gates.

For instance, for the rules used previously:

```
/* R1 */ goal1 :- fact1.
/* R2 */ goal1 :- a, b.
/* R3 */ goal2(X) :- c(X).
/* R4 */ a :- not(d).
/* R5 */ b :- d.
/* R6 */ b :- e.
/* R7 */ c(2) :- not(e).
/* R8 */ d :- fact2, fact3.
/* R9 */ e :- fact2, fact4.
```

the and-or-not lattice is shown in Figure 6-7. Here we follow the usual conventions for gate shapes (see Appendix A); facts are the far-left-side "inputs" and goals are the far-right-side "outputs" of this logic network. Truth of a fact or conclusion is represented by a high or "on" voltage along a line, falsity by a low or "off" voltage. The output (right-side) voltage of a gate is determined by the logic function of the voltages representing input(s) on the left side; "and" gates determine a logical "and", "or" gates logical "or", and inverter gates logical "not". A given set of facts sets the input voltages for a group of initial gates, which determine voltages of others, and so on, with everything proceeding in parallel. The and-or-not lattice is a useful interpretation of rules and facts for the many applications in which order within "and"s and "or"s doesn't mean anything. It also provides a specification for an integrated-circuit chip that could do this very fast.

This unordered interpretation of rules takes us to the opposite extreme of the classical Eckert-Mauchly | REFERENCE 2| model of the computer as a sequential processor. .FS | REFERENCE 2| Sometimes called the Von Neumann model, but evidence now suggests that Von Neumann had little to do with it. .FE The main advantage of the and-or-not lattice is processing speed. Another advantage is the partitionability into modules: we can identify subnetworks and their inputs and outputs, and then build them easily into large networks. We don't need to worry much about *redundancies*, gates whose effect is also accomplished by other gates; they can't slow computation. But it is true we can't have contradictory gates or results will be meaningless.

Like decision lattices, and-or-not lattices do have the disadvantage that they can't handle variables well (you'll notice the example has none). Gates can't directly "bind" variables, though we could indirectly represent alternative bindings by having multi-position switches on each input to the logic network, switches that could connect the inputs to one of several lines representing the truths of different facts with the same predicate name. Then binding means selecting a switch position, and trying possible bindings means turning the switch and watching the conclusions reached. But this is awkward, and--especially when there is more than one variable--can greatly increase the time to get an answer, a main reason for using and-or-not lattices in the first place.

Randomness in control structures

A control structure need not be deterministic (that is, always work the same way on the same rules and facts). Human beings seem to have some randomness in their actions. So a control structure can make some random choices, especially when alternatives seem of equal worth. For instance, maybe in backward chaining when there are more than ten rules applicable to a situation, choose one at random. Randomness is most appropriate for rule-based systems trying to model skilled real-time performance by people. Randomness can prevent infinite loops, because when you choose randomly you can't get stuck in a rut.

Grammars for interpreting languages (*)

Human and computer languages can be handled and processed with *grammars*. Grammars are rule-based systems different from those considered so far. For one thing, grammar rules don't deal with facts, but with strings of words or symbols in a particular order. All grammar rules are of a particular narrow form: a statement that one string of words can be substituted for another string of words. Though we didn't say so, we saw an example of a grammar in Section 5.9: those substitution pairs, though disguised as facts. But grammars can also have *nonterminal* words, words that symbolize grammatical categories and are not part of the language itself. For instance, the nonterminal "noun" can be substituted for the word "man" or the word "computer". Nonterminals can also describe sequences of words, like identifying a "noun phrase" as a determiner followed by an adjective followed by a noun. Linguists have lots of these categories.

Grammars are essential for getting computers to understand sentences in a language, because they make it possible to determine the structure of a sentence. Used this way, grammars are analogous to the rule-based systems in this chapter. The "facts" are the words of a sentence, and the goal is to change that sentence, by substitutions, into the sentence consisting of the single nonterminal called "sentence". Such a process is analogous to forward chaining, and is called *bottom-up parsing*. It usually involves substituting shorter sequences of words for longer sequences of words. But you can also work in the opposite direction from "sentence" to a list of actual language words, analogously to backward chaining, and this is called *top-down parsing*. It usually involves substituting longer sequences of words for shorter sequences of words, and so is likely to require more backtracking than bottom-up parsing, but less work per backtrack since shorter sequences are searched for. Figure 6-8 shows an example; upward arrows represent bottom-up parsing, downward arrows top-down.

Hybrids that compromise on the advantages and disadvantages of both bottom-up and top-down parsing are possible. Grammar rule order is important for efficiency: the most likely rules should be tried first. Parts of the sentence can be processed concurrently. Partitioning can be used to group information about related words and related parsing rules into modules. Meta-rules can be used for sophisticated control more like what people do.

Today most artificial-intelligence work in understanding sentences from human languages (or *natural languages*) uses a variation on top-down parsing called *augmented transition networks* (ATNs). These are a "smarter" kind of top-down parsing that attaches additional conditions to the parsing rules, so that only the most "reasonable" rules that apply will be tried. They also use recursion in a complicated way similar to the one in Chapter 11.

Keywords:

rule-based systems
control structure
conflict resolution
expert systems
compiled control structures
backward chaining
goal
virtual facts
the asserta predicate
the retract predicate
forward chaining
focus-of-attention feature
rule-cycle hybrid control structure
rule and fact priorities
rule specificity
partitioned control structures
meta-rules
decision trees
partition parallelism
"or" parallelism
demons
"and" parallelism
variable-matching parallelism
object-oriented programming
and-or-not lattice
grammar
top-down parsing
bottom-up parsing
augmented transition networks (ATNs)

Exercises

6-1. (A) Here's a rule-based system:

```
/* R1 */ k(X) :- j(X), b(X).
/* R2 */ f(X) :- a(X), not(g(X)).
/* R3 */ a(X) :- b(X), i.
/* R4 */ d :- i.
/* R5 */ d :- e(X), c.
/* R6 */ g(X) :- h, a(X).
/* R7 */ g(X) :- l.
/* R8 */ b(X) :- c.
```

Assume the goals are f(X), d, and k(X), in that order, and the facts are c, **l**, **e(a)**, and **j(b)**, in that order. Assume no extra caching. Assume we stop when a goal is proved.

(a) Suppose we use pure forward chaining. List the rule invocations, successes, and failures, in order as they occur. Use the rule numbers to indicate rules.

(b) Now list the rule invocations, successes, and failures for backward chaining, in order.

(c) Does fact order affect which goal is proved first with forward chaining? Why?

(d) Does fact order affect which goal is proved first with backward chaining here? Why?

6-2. Consider this database:

```
top(X,Y,Z) :- bottom(Z,W,Y,X).
bottom(A,B,7,D) :- data(A,0,B), data(A,D,1).
data(3,0,1).
data(3,2,1).
```

List in order the facts that are proved by pure forward chaining using focus-of-attention placement of new facts. Don't stop until everything provable is proved.

6-3. Suppose we are doing pure forward chaining on a set of R rules, rules without semicolons ("or"s), nots, arithmetic, and variables. Suppose these rules have L distinct predicate expressions on the left sides, and S distinct predicate expressions, each occurring no more than M times, on the right sides. Suppose there are T total predicate expressions on right sides. Suppose there are F facts, F>0.

(a) What is the maximum number of locations immediately matchable to facts (that is, without having to prove new facts) on rule right sides?

(b) What is the maximum number of new facts that can be eventually found by forward chaining?

(c) What would be the effect on your answer to part (a) of allowing one-argument variables in rules and facts?

(d) What would be the effect on your answer to part (b) of allowing one-argument variables in rules and facts? (A conclusion with an unbound variable still counts as one new fact.)

6-4. Consider the following Prolog database:

```
a(X) :- b(X).
b(X) :- c(X), d(X).
d(X) :- e, f(X).
c(X) :- g(X).
g(2).
f(5).
g(5).
e.
```

(a) What new facts are proved in order by the rule-cycle hybrid of forward and backward chaining? Continue until all possible facts are found.

(b) What new facts are proved in order by the pure form of forward chaining with focus-of-attention conflict resolution for facts? Continue until all possible facts are found.

6-5. (R,A) Consider the following rules and facts:

```
a :- v, t.
a :- b, u, not(t).
m(X) :- n(X), b.
b :- c.
t :- r, s.
u :- v, r.
r.
v.
c.
n(12).
```

(a) Suppose we do pure forward chaining with focus-of-attention placement of new facts. Suppose we want all possible conclusions. List in order the new facts derived from the preceding. Remember we must save nots for last.

(b) Suppose we do rule-cycle hybrid chaining with focus-of-attention placement of new facts, saving nots for last, and we want all possible conclusions. List in order the new facts derived.

(c) One problem with rule-cycle hybrid chaining is that it repeatedly checks the same rules again on each cycle. Describe a way to know when not to check a rule because of what happened on the last cycle, besides the idea of deleting a rule without variables when it succeeds. (Hint: using this idea you only need check nine rules total for the preceding rules and facts.)

6-6. (E) Consider a variant of the rule-cycle hybrid that only cycles through the rules once. How is this like an assembly line in a factory?

6-7. (A) Suppose we are doing rule-cycle hybrid chaining on R rules, rules without variables and nots. Suppose there are L distinct predicate names on rule left sides, and S distinct predicate names on rule right sides out of T total predicate names on right sides. Suppose there are F facts, F>0. What is the maximum number of cycles needed to prove everything that can be proved?

6-8. (A,H) Reasoning about rules and facts written in English instead of Prolog can be tricky, and you must be careful. Consider the following rule-based system for actions of a small reconnaissance robot. Suppose the following facts are true, in this order:

> **F1: There is an object with branches to the right of you.**
> **F2: This object is 2 feet tall.**
> **F3: This object occupies 20 cubic feet.**
> **F4: This object is stationary.**
> **F5: Another object is moving towards you.**
> **F6: You hear speech from that object. (Assume speech is not a loud noise.)**

Assume all other facts mentioned in rules are false. Assume any new facts, when added, will be put at the front of the list of facts. Assume the rule-based system can result in the following actions, in this order:

> **A1: Turn around.**
> **A2: Stop and wait.**
> **A3: Turn towards something.**
> **A4: Move a short distance forward.**
> **A5: Turn 20 degrees right.**
> **A6: Move a long distance forward.**

Here are the rules:

> **R1: If you hear a loud noise in front of you, then turn around and move a long distance.**
>
> **R2: If you want to hide, and there is a bush nearby, then turn towards the bush, and**

move a short distance.

R3: If you want to hide, and are beneath a bush, then stop and wait.

R4: If an object is moving towards you, and it is a person or vehicle, then hide.

R5: If an object is moving, and it is an animal, then stop and wait.

R6: If an object is an obstacle and you are moving, and the object is blocking your path, then turn right 20 degrees, and move a short distance.

R7: Move forward a long distance. [notice no "if" part here]

R8: If an object has long branches, and the branches are moving, and it does not have wheels, then it is an animal.

R9: If an object makes irregular noises, then it is an animal.

R10: If an object makes regular noises, and it is moving, it is a vehicle.

R11: If an object has wheels, then it is a vehicle.

R12: If an object is stationary, and occupies more than 1 cubic foot, it is an obstacle.

R13: If an obstacle has branches, and is less than 3 feet tall, it is a bush.

R14: If an obstacle has branches, and is more than 3 feet tall, it is a tree.

R15: If an obstacle has no branches, it is a rock.

R16: If an animal has four branches in two pairs, and one pair supports the animal, it is a person.

R17: If an animal speaks, it is a person.

(a) List in order the rule invocations, successes, and failures with backward chaining. Assume conflict resolution based on the rule order given. Assume caching of proved facts, so once something is concluded it need never be figured out again.

(b) List in order the rules invoked with forward chaining, ignoring rule R7. Again, take rules in the order given,

(c) Give a different kind of conflict resolution that would work well for this rule-based system.

6-9. (A) A rule-based system is monotonic if anything that can be concluded at one time can be concluded at any later time. Consider a consistent (that is, non-self-contradictory) "pure" backward-chaining rule-based system, one that doesn't use asserta or retract or any built-in predicates besides not. Such a system is necessarily monotonic.

(a) Suppose the system is partitioned into modules whose rules cannot "see" the rules of other modules until those modules are specifically loaded. Must the new scheme be monotonic?

(b) Suppose we cache intermediate and final conclusions reached by the system, using asserta. Must the new scheme be monotonic?

(c) Suppose we add to right sides of rules in the system asserta predicate expressions with arbitrary arguments. Must the new scheme be monotonic?

6-10. (E) (a) Despite their similar names, decision lattices and and-or-not lattices are quite different things. List their major differences.

(b) List their major similarities.

6-11. Consider the following rule-based system for self-diagnosis and treatment of colds and flu:

> **A. To sleep, go to bedroom.**
> **B. To drink fluids, go to kitchen.**
> **C. To drink fluids, go to bathroom.**
> **D. To take temperature, go to bathroom.**
> **E. To take aspirin, go to bathroom.**
> **F. To telephone, go to living room.**
> **G. To telephone, go to kitchen.**
> **H. If feel headache or nasal congestion, then feel sick.**
> **I. If feel sick, then take temperature.**
> **J. If have fever, then take aspirin.**
> **K. If have fever, then call boss.**
> **L. If have fever, then go to bed.**
> **M. If have nasal congestion or fever, then drink fluids.**

(a) Order the rules in a good way. Show how backward chaining would work applied to the circumstance when you wake up with a headache, nasal congestion, and a fever. (Use the letter in front of each rule to identify it.)

(b) Suggest a fundamentally different control structure than the preceding, one good for this problem. (Find a general control structure that could work in many daily-living rule-based systems, not just this problem.) Show how it would work on the first eight rules applied to the same situation.

6-12. (R,A) Consider using a rule-based system as the brains of a "smart house", a house that automatically does a lot of things that people do for themselves in other houses. Assume the smart house can control power to all the electrical sockets, so it can turn lights, heat and appliances on and off at programmed times. The smart house can also monitor many different kinds of sensors--for instance, light sensors to turn off outside lights during the daytime, infared sensors to detect when people are in a room, audio sensors to detect unusual noises, and contact sensors to detect window openings (as by a burglar). Sensors permit flexible, unprogrammed action, like sounding a burglar alarm in the master bedroom only if someone is there, or turning off appliances when they seem to have been left on accidentally and no one is in the room anymore. Priorities could be established, so routine activities won't be allowed when sensors say a fire is progress. Assume the rule-based system is invoked repeatedly to check for new conditions.

(a) Which is better for this problem, backward chaining or forward chaining? Why?

(b) Is caching a good idea? Why? If so, how would it work?

(c) Are virtual facts a good idea? Why? If so, how would it work?

(d) Is rule partitioning a good idea? Why? If so, how would it work?

(e) Are decision lattices a good idea? Why? If so, how would they work?

(f) Are and-or-not lattices implemented on integrated circuits a good idea? Why? If so, how would they work?

6-13. On the popular fictitious television show *The Citizenry's Courtroom*, Judge Wimpner hears small-claims cases and makes monetary awards of varying amounts to the plaintiff. But the award is misleading because the plaintiff and defendant are also paid for their appearance on the program, as one will discover on reading the small print of a disclaimer flashed on the screen for a second during the closing credits. The plaintiff is given 25 dollars plus the amount of the award plus half of a pool. The defendant is given 25 dollars plus the other half of the pool. The pool is zero if the amount of the award is 500 dollars or more, otherwise the difference between 500 dollars and the amount of the award.

(a) Suppose the award amount is represented in Prolog as a fact with predicate name **award** and one argument representing the amount of the award in dollars. Write three Prolog rules representing the stated calculations necessary for the plaintiff's amount, the defendant's amount, and the pool amount. Don't simplify; represent exactly what was stated.

(b) Now "compile" the three preceding rules into two rules for the plaintiff amount and the defendant amount. Refer to the award amount only once in each rule.

(c) Discuss what is needed to do this compilation for any set of rules involving arithmetic. Will a few simple tricks do it, or is a large rule-based expert system of its own necessary?

(d) What are the disadvantages of this form of compilation?

(e) Compare and contrast this kind of compilation with and-or-not lattices.

6-14. (E) *Occam's Razor* is the principle that when several alternative explanations are possible for something, you should pick the simplest one. Suppose we try to apply Occam's Razor to a rule-based system in which the left sides of rules represent explanations of phenomena.

(a) Which of the terms in Figure 6-1 best describes Occam's Razor?

(b) Why would it be very difficult to apply Occam's Razor to a practical problem?

6-15. (E) Compare and contrast the control structures discussed in this chapter with control of activities (by directive) in bureaucracies.

6-16. (P) Write a Prolog program to drive a car. Assume the following sensor facts are known to you, as "data":

```
S1: you are traveling below the speed limit
S2: you are traveling at the speed limit
S3: you are traveling above the speed limit
S4: you are on a two-lane road
S5: you are on a four-lane road
S6: you see an intersection coming up
S7: a car is less than 100 meters in front of you
S8: the road changes from two-lane to four-lane shortly
S9: the road will change from four-lane to two-lane shortly
S10: the brake lights of the car in front of you are on
S11: you are getting closer to the car in front of you
S12: you are passing another car going in the same direction
```

Assume the only actions are

```
A1: speed up
A2: slow down
```

```
A3: maintain speed
A4: pass a car in front of you (or keep passing one you're passing)
```

Here are examples of what you want the program to do. (These are not good rules--they're too specific.)

```
S2,S4,S6: slow down (A2)
S2,S5,S8,S10: slow down (A2)
S1,S5,S11: pass (A4)
S1,S5: speed up (A1)
S3,S4,S7: slow down (A2)
```

Write a Prolog program that decides what to do for every possible combination of the sensor facts. (Since there are 12 kinds, you must handle |2 sup 12|, or 4096 different situations). Assume your program is called every second to decide what to do that second. Choose a good control structure, and discuss (justify) your ordering of rules. Try to drive safely.

6-17. (E) Sometimes the ideas of science fiction writers seem pretty wild, but sometimes the ideas that seem wild really aren't when you think about them a bit.

(a) In *Camp Concentration* by Thomas Disch (1968), it's suggested that a disease that rearranges nerve patterns in the human brain could cause people to become a lot smarter, more able to reason and make inferences. Why is this not reasonable based on the assumption that people reason using the methods of this chapter? Suggest a small change to ideas from this chapter that would permit this phenomenon.

(b) In *Brain Wave* by Poul Anderson (l954), it's suggested that a small increase in speed of the nerve impulses in the human brain could lead to a much larger increase in the speed at which humans reason and make inferences. Why is this not reasonable based on the assumption that people reason using the methods of this chapter? Give a small change to ideas from this chapter that permit this phenomenon.

Go to book index

Implementation of rule-based systems

Let's examine how different control structures for rule-based systems can be implemented in Prolog. We'll cover backward chaining, rule-cycle hybrid chaining, forward chaining, input and output routines, meta-rules, and decision lattices. This will mean a lot of details; the last section of this chapter brings together the key Prolog code.

Implementing backward chaining

Though Prolog is designed for backward chaining, there are many details for an implementer of a backward-chaining rule-based system to worry about when using Prolog, especially when the rule-based system is large. The traffic lights program in Section 4.11 was simple because traffic laws are *supposed* to be simple, so even people unenlightened enough never to have taken an artificial intelligence course can understand them. Many rule-based expert systems can't be so nice, like those that diagnose and fix hard problems. Such rule-based systems are often called *expert systems*, because they automate the role of human experts. We'll introduce some general-purpose programming aids for expert systems in this chapter.

The goal of most expert systems is to reach a diagnosis, which we'll assume is obtained by typing the query

```
?- diagnosis(X).
```

So what is X? It should describe a situation. We could connect words with underscores as before, but there's an alternative: we can put single quotation marks (apostrophes) around a string of words to force treatment of it as a unit. That is, a *character string*. An advantage of character strings is that they can start with capital letters and contain periods and commas, while words can't.

Here are some example diagnosis rules for an expert system:

```
diagnosis('fuse blown') :- doesnt_work, all_lights_out.
diagnosis('fuse blown') :- noise(pop).
diagnosis('break in cord') :- doesnt_work, cord_frayed.
```

Of course we must define those right-side predicates.

Implementing virtual facts in caching

One problem is those rules require advance entry of facts (often, many facts) so that rule right sides can work properly. As we mentioned in the last chapter, virtual facts (facts demanded only when needed) are a simple improvement. A good way to get them is to define a function predicate **ask** of two bound arguments. The first argument is an input, a string containing question text to be typed out on the terminal, and the second argument is an output, a variable to be bound to the question's answer that the user types.

```
ask(Q,A) :- write(Q), write('?'), read(A), nl.
```

Here **write**, **read**, and **nl** are Prolog predicates built-in in most implementations (see Appendix D); **write** prints its argument on the terminal, **read** reads something typed by the user and binds that something to the variable that is the **read**'s argument, and **nl** sends a carriage return to the terminal.

Now we never want to ask a user the same question twice; we should cache answers so we can reuse them. It's easy to add this feature to the **ask** predicate. We just use the **asserta** built-in predicate introduced in Section 6.1, which takes a fact as argument and adds it to the Prolog database. Using it, conclusions can be added to the database as they are discovered. We can stick the **asserta** at the end of the definition of the **ask** predicate:

```
ask(Q,A) :- write(Q), write('?'), read(A), nl, asserta(ask(Q,A)).
```

Then if the same question is asked again, the fact will be used to answer it instead of this rule. This works because facts put in the database with **asserta** are put in front of all other facts and rules with the same first predicate name. Here's an example of the use of **ask**:

```
diagnosis('fuse blown') :- ask('Does the device work at all',no),
  ask(Are the lights in the house off',yes).
```

This says to diagnose that the fuse is blown if (1) the user answers **no** when asked whether the device works at all, and (2) the user answers **yes** when asked whether all the lights in the house are off.

You should carefully phrase the questions to be issued by a rule-based system. In particular, avoid pronouns and other indirect references to things, since rules and questions may be invoked in hard-to-predict orders. Generally, though there are exceptions, phrase questions so a **yes** answer means unusual things are going on, while a **no** means things are normal. For instance, after "Are the lights in the house off?", don't ask "Is the fuse OK?" but "Is the fuse blown?". And be consistent in phrasing questions. After that question "Are the lights in the house off?", ask "Does the fuse look blown?" in preference to than "The fuse looks blown doesn't it?", to maintain the same verb-noun-adjective order.

Input coding

We could implement a big expert system this way, with diagnosis rules having **ask** predicates on their right sides. But this can require unnecessary code repetition. So two important coding tricks are used in large rule-based systems: coding of input (answers) and coding of output (questions and diagnoses).

Input coding groups user answers into categories. An important case is questions with only **yes** or **no** answers; expert systems often to rely on them for simplicity. We can define two new predicates **affirmative** and **negative**, which say whether a word the user typed is a positive or a negative answer respectively:

```
affirmative(yes).
affirmative(y).
affirmative(ye).
affirmative(right).
affirmative(ok).
affirmative(uhhuh).
negative(no).
negative(n).
negative(not).
negative(never).
negative(impossible).
negative(haha).
```

Then we can define a predicate **askif** of one input argument. It will be just like **ask** except it will have only one argument, the question, and it will succeed if that question is answered affirmatively and fail if the question is answered negatively. We can also fix it so that if an answer is neither positive nor negative (in other words, it is unclear), we will complain and ask for another answer.

```
askif(Q) :- ask(Q,A), positive_answer(A).
positive_answer(A) :- affirmative(A).
positive_answer(Qcode,A) :- not(negative(A)), not(affirmative(A)),
  write('Please answer yes or no.'), read(A2),
  retract(asked(Qcode,A)), asserta(asked(Qcode,A2)),
  affirmative(A2).
```

We can also define:

```
askifnot(Q) :- not(askif(Q)).
```

which saves some parentheses.

Users may not always understand a question. We can let them type a **?** instead of an answer, give them some explanatory text, and provide them another chance to answer:

```
ask(Q,A) :- asked(Q,A).
ask(Q,A) :- not(asked(Q,A)), write(Q), write('? '), read(A2),
  ask2(Q,A2,A).
ask2(Q,'?',A) :- explain(Q), ask(Q,A).
ask2(Q,A,A) :- not(A='?'), nl, asserta(asked(Q,A)).
```

where **explain** facts store explanatory text. Minor humanizing touches such as these can be immensely important to user satisfaction, while imposing little on the programmer.

Output coding

Another useful trick is to code questions, so we need not repeat their text at each mention in the rules. Codes for questions also make rules easier to read, and help prevent mistakes because it's easy to err in typing a long string of text (and with caching, every slightly different question is asked and cached separately). For this we can use a predicate **questioncode** of two arguments, a code word and a text string for the corresponding question. Here's an example from appliance diagnosis:

```
diagnosis('fuse blown') :- askif(device_dead), askif(lights_out).
diagnosis('fuse blown') :- askif(hear_pop).
diagnosis('break in cord') :- askif(device_dead),
  askif(cord_frayed).
questioncode(device_dead,'Does the device refuse to do anything').
questioncode(lights_out,
  'Do all the lights in the house seem to be off').
questioncode(hear_pop,'Did you hear a sound like a pop').
questioncode(cord_frayed,
  'Does the outer covering of the cord appear to be coming apart').
```

To handle this, we must redefine **ask**:

```
ask(Qcode,A) :- asked(Qcode,A).
ask(Qcode,A) :- not(asked(Qcode,A)), questioncode(Qcode,Q),
  write(Q), write('? '), read(A2), ask2(Q,Qcode,A2,A).
ask2(Q,Qcode,'?',A) :- explain(Qcode), ask(Qcode,A).
ask2(Q,Qcode,A,A) :- not(A='?'), asserta(asked(Qcode,A)).
```

A further refinement is to handle a class of related questions together. We can do this by giving arguments to output codes, as for instance using **hear(X)** to represent a question about hearing a sound **X**. Then to make the query we need string concatenation, something unfortunately not available in most Prolog dialects. But there is a simple shortcut to concatenation by writing a **questioncode** rule instead of a fact, that types extra words before succeeding:

```
questioncode(hear(X),X) :- write('Did you hear a sound like a ').
```

So to ask if the user heard a pop sound you use:

```
askif(hear(pop)).
```

which prints on the terminal as:

```
Did you hear a sound like a pop ?
```

Yet another coding trick is to code diagnoses. This isn't as useful as question coding, but helps when diagnoses are provable in many different ways. Diagnosis coding requires a new top-level predicate that users must query instead of **diagnosis**, as:

```
coded_diagnosis(D) :- diagnosis(X), diagnosis_code(X,D).
```

For instance, we could use:

```
diagnosis(fuse) :- ask('Does the device work at all',no),
  ask(Are the lights in the house off',yes).
diagnosis_code(fuse,'Fuse blown').
```

Then we could get this behavior:

```
?- coded_diagnosis(X).
Does the device work at all? no.
Are the lights in the house off? yes.
X=Fuse blown
```

Intermediate predicates

Building expert systems is straightforward when there are ten to a hundred rules, each with one to three expressions on the right side. But it can get confusing when, as in a typical expert system today, there are thousands of rules averaging ten expressions per right side. It just gets too difficult to keep track of all the symbols used, and to determine everything necessary for each rule. The solution is to frequently use intermediate predicates, predicates that occur on both the left and right sides of rules. Intermediate predicates can represent important simplifying generalizations about groups of facts. Viewing expert-system predicates as a hierarchy with diagnoses or final conclusions at the top and asked questions at the bottom, intermediate predicates are everything in between. Much of the intelligence and sophistication of expert systems can come from a good choice of intermediate predicates to reduce redundancy and simplify rules.

A useful intermediate predicate with appliance diagnosis is **power_problem**, a predicate that is true if the appliance is not getting any electricity for its innards. It's useful because appropriate diagnoses when it is true are quite different from those when it is false: a non-power problem must be in the device itself and can't be in the cord or external fuse. So **power_problem** can go into many rules, typically early in the right side of the rule to filter out inappropriate rule use. But **power_problem** clearly is an intermediate predicate, not a fact we can establish from a single question, because it has many different symptoms. And some symptoms are very strong, as when all the lights in the house went off when you have tried to turn on the device, or when the device stopped working when you moved its frayed cord slightly.

Generally speaking, intermediate predicates are needed for any important phenomena that aren't diagnoses. Here are some more ideas for intermediate predicates in appliance diagnosis:

--whether the problem is mechanical;
--whether the problem is in a heating element;
--whether the appliance has had similar problems before;
--whether you can improve things by adjusting the controls or buttons;
--whether danger of electrocution is present;

--whether anything unusual was done to the appliance lately (like being dropped or having liquids spilled on it);
--how much troubleshooting expertise the user has (note that intermediate predicates can have arguments).

Intermediate-predicate expressions won't be arguments to **askif**s, since they don't directly query a user. Caching is a good idea with intermediate predicates, even more so than caching of query answers (as we did with the **ask** predicate). A single intermediate-predicate fact can summarize many questions, and caching it saves having to ask all those questions over again. Nevertheless, caching of intermediate-predicate conclusions should not necessarily be automatic, as it only makes sense when a result might be reused.

An example program

Let's put together the ideas we've introduced in this chapter in a larger rule-based system for the diagnosis of malfunctions in small household appliances. Figure 7-1 shows some of the terminology, and Figure 7-2 gives the predicate hierarchy. We list rules in three groups: diagnosis (top-level) rules, intermediate predicate rules, and question-decoding rules. To get a diagnosis from this program, query **diagnosis(X)**. Typing a semicolon will then give you an alternative diagnosis, if any; and so on. So if several things are wrong with the appliance, the program will eventually find them all.

```
/* Top-level diagnosis rules */
diagnosis('fuse blown') :- power_problem, askif(lights_out).
diagnosis('fuse blown') :- power_problem, askif(hear(pop)).
diagnosis('break in cord') :- power_problem, askif(cord_frayed).
diagnosis('short in cord') :- diagnosis('fuse blown'),
  askif(cord_frayed).
diagnosis('device not turned on') :- power_problem,
  klutz_user, askif(has('an on-off switch or control')),
  askif(device_on).
diagnosis('cord not in socket properly') :- power_problem,
  klutz_user, askif(just_plugged), askif(in_socket).
diagnosis('foreign matter caught on heating element') :-
  heating_element, not(power_problem), askif(smell_smoke).
diagnosis('appliance wet--dry it out and try again') :-
  power_problem, klutz_user, askif(liquids).
diagnosis('controls adjusted improperly') :- klutz_user,
  askif(has('knobs, switches, or other controls')).
diagnosis('kick it, then try it again') :- mechanical_problem.
diagnosis('throw it out and get a new one') :-
  not(power_problem), askif(hear('weird noise')).
diagnosis('throw it out and get a new one').

/* Definitions of intermediate predicates */
power_problem :- askif(device_dead), askif(has(knobs)),
askif(knobs_do_something).
```

```
power_problem :- askif(device_dead), askif(smell_smoke).
klutz_user :- askifnot(handyperson).
klutz_user :- askifnot(familiar_appliance).
mechanical_problem :- askif(hear('weird noise')),
  askif(has('moving parts')).
heating_element :- askif(heats).
heating_element :- askif(powerful).

/* Question decoding */
questioncode(device_dead,'Does the device refuse to do anything').
questioncode(knobs_do_something,
  'Does changing the switch positions or turning
  the knobs change anything').
questioncode(lights_out,
  'Do all the lights in the house seem to be off').
questioncode(cord_frayed,
  'Does the outer covering of the cord appear to be coming apart').
questioncode(handyperson,'Are you good at fixing things').
questioncode(familiar_appliance,
  'Are you familiar with how this appliance works').
questioncode(device_on,'Is the ON/OFF switch set to ON').
questioncode(just_plugged,'Did you just plug the appliance in').
questioncode(in_socket,'Is the cord firmly plugged into the socket').
questioncode(smell_smoke,'Do you smell smoke').
questioncode(liquids,
  'Have any liquids spilled on the appliance just now').
questioncode(heats,'Does the appliance heat things').
questioncode(powerful,'Does the appliance require a lot of power').
questioncode(has(X),X) :- write('Does the appliance have').
questioncode(hear(X),X) :- write('Did you hear a ').
```

Here we use variables inside question codes for questions about components and sounds heard. We also use a *subdiagnosis* ('fuse blown') as input to another diagnosis, a useful trick.

Running the example program

Here's an actual run of this program (which requires definitions of **askif** and other predicates given earlier in this chapter). Note the same diagnosis is repeated when there are different ways to prove it.

```
?- diagnosis(X).
Does the device refuse to do anything? yes.
Do all the lights in the house seem to be off? no.
Does the appliance have knobs or switches? yes.
Does changing the switch positions or turning
the knobs change anything? no.
```

```
Do you smell smoke? yes.
Does the appliance heat things? no.
Does the appliance require a lot of power? no.
Did you hear a pop? yes.


X=Fuse blown;

X=Fuse blown;

X=Fuse blown;
Are you good at fixing things? no.
Does the appliance have an on-off switch or control? yes.
Is the ON/OFF switch set to ON? no.

X=Device not turned on;

Are you familiar with how this appliance works? no.

X=Device not turned on;

X=Device not turned on;

X=Device not turned on;

X=Device not turned on;

X=Device not turned on;

Did you just plug the appliance in? yes.
Is the cord firmly plugged into the socket? no.

X=Cord not in socket properly;

X=Cord not in socket properly;

X=Cord not in socket properly;

X=Cord not in socket properly;

X=Cord not in socket properly;

X=Cord not in socket properly;

Have any liquids spilled on the appliance just now? maybe.
Please type yes or no. no.

X=Controls adjusted improperly;
```

```
X=Controls adjusted improperly;
Did you hear a weird noise? no.

X=Throw it out and get a new one;

no
?- halt.
```

Partitioned rule-based systems

Intermediate predicates group related rules together, but they are only a conceptual grouping, more a help to understanding and debugging programs. A stronger way of grouping is putting rules into partitions that can't "see" one another. This is easy to do with Prolog by putting rules in separate files and only loading the files you need into the database. Loading is done with the **consult** built-in predicate in Prolog, a predicate of one argument which is the name of the file to load. So if the rule

```
a :- b, c.
```

is used for backward chaining, and we want whenever it succeeds for the file "more" to be loaded, we should rewrite it as

```
a :- b, c, consult(more).
```

Like most built-in predicates, **consult** always fails on backtracking since there's only one way to load a file.

Often one partition is designated the "starting" partition, loaded automatically when the rule-based system begins. It then decides which other partitions to load and invoke. If a loaded partition later decides it's not relevant (as when none of its rules fire), it can itself load another partition and start that one running.

Implementing the rule-cycle hybrid

Prolog's features make backward chaining easy. But it's also a general-purpose programming language, and can implement quite different control structures.

First consider the rule-cycle hybrid of backward and forward chaining, easier to implement than pure forward chaining and hence used in many simple expert systems. It can be done by writing each rule in a new form, a transformation of each backward chaining rule:

1. "and" a new **asserta** on the right end of the right side of the rule, whose argument is the left side of the rule;

2. "and" a new **not** on the left end of the right side of the rule, with the same argument;

3. then replace the left side of the rule by **r** (first renaming any **r** predicates already in the rules).

So these backward chaining rules:

```
a :- b.
c :- d, e, f.
```

become:

```
r :- not(a), b, asserta(a).
r :- not(c), d, e, f, asserta(c).
```

And two sample rules from the appliance diagnosis program:

```
diagnosis('fuse blown') :- power_problem, askif(lights_out).
power_problem :- askif(device_dead), askif(has(knobs)),
  askif(knobs_do_something).
```

become:

```
r :- not(diagnosis('fuse blown')), power_problem, askif(lights_out),
  asserta(diagnosis('fuse blown')).
r :- not(power_problem), askif(device_dead), askif(has(knobs)),
  askif(knobs_do_something), asserta(power_problem).
```

(We'll discuss later how to convert automatically.) So we replace our old rules with new rules whose only effect is caching of particular conclusions. Note that these new rules never call on other rules, even if there are intermediate predicates, because **r** is the only left side (since we made sure the predicate **r** doesn't occur in the rule-based system). We'll assume for now that no predicate expressions in the original rules contain **not**s, since they introduce complications. If we really need negatives, we can define fact predicates that stand for the opposite of other fact predicates.

Now we must cycle through the rules; that is, consider each rule in order, and go back to the first when we finish the last. Within each pass, we can force the Prolog interpreter to repeatedly backtrack to a query of **r**. A simple way is

```
?- r, 1=2.
```

Since the = can never be true, the interpreter will keep trying **r** rules, regardless of whether they succeed or fail. Eventually it will run out of all **r** rules and fail. Actually, there's a built-in Prolog predicate called **fail** that has exactly the same effect as **1=2**, so we can say equivalently

```
?- r, fail.
```

To give us a handle on this code, let's give it a name:

```
one_cycle :- r, fail.
```

To get the Prolog interpreter to repeat indefinitely a cycle through the rules, we might think that we could do the same **fail** trick, like

```
hybrid :- one_cycle, fail.
```

But this won't work because **one_cycle** won't ever return to the *top* of the list of rules. And **one_cycle** itself never succeeds, so the **fail** is useless. We could try

```
hybrid :- not(one_cycle), fail.
```

which answers the second objection but not the first: we need each call to **one_cycle** to be a fresh call. That suggests recursion:

```
hybrid :- done.
hybrid :- not(one_cycle), hybrid.
```

The **done** is a stopping condition that must be defined by the builder of the rule-based system. For diagnosis expert systems, it could be defined as

```
done :- diagnosis(X).
```

which says to stop whenever some diagnosis is proved.

The preceding definition of **hybrid** only checks once per cycle whether it is done. To stop sooner, we could put the check inside **one_cycle** like this:

```
hybrid :- done.
hybrid :- not(one_cycle), hybrid.
one_cycle :- r, done.
```

But this requires more calls to **done**, not a good idea if **done** is a complicated calculation.

Note: this approach can handle **not**s in rules, though differently from the algorithm in Section 6.4 since **not**s will be evaluated on every cycle. But as with the algorithm, any **not** must occur before any rule with the argument to the **not** as its left side, or we'll get wrong answers.

Implementing pure forward chaining (*)

Pure forward chaining requires yet another rule form. (See a Section 7.14 for how to rewrite rules automatically in this form.) Since pure forward chaining repeatedly finds and "crosses out" expressions on the right sides of rules, it would help to express rule right sides as lists, for then we can use our **member** and **delete** list-processing predicates from Chapter 5. We can do this by making rules a kind of fact, say using a **rule** predicate name. The first argument to **rule** can be the left side of the original rule, and the second argument the list of predicate expressions "and"ed on the right side. So these rules

```
a :- b.
c :- d, e, f.
g(X) :- h(X,Y), not(f(Y)).
```

become

```
rule(a,[b]).
rule(c,[d,e,f]).
rule(g(X),[h(X,Y),not(f(Y))]).
```

and the two sample rules from the appliance diagnosis program

```
diagnosis('fuse blown') :- power_problem, askif(lights_out).
power_problem :- askif(device_dead), askif(has(knobs)),
  askif(knobs_do_something).
```

become

```
rule(diagnosis('fuse blown'),[power_problem,askif(lights_out)]).
rule(power_problem,
  [askif(device_dead),askif(has(knobs)),askif(knobs_do_something)]).
```

For now, we'll assume that the rules don't contain **not**s.

We also must represent facts. Pure forward chaining requires that we identify all facts, distinguishing them from rules. We can do this by making each fact an argument to a predicate named **fact**, of one argument. Then to bind **F** to every fact in turn, we query

```
?- fact(F), done.
```

which will backtrack repeatedly into **fact**. For every fact **F**, we must find the rules whose right sides can match it, to derive new rules and possibly new facts. This suggests:

```
forward :- fact(F), not(pursuit(F)), done.
```

Unfortunately, we can't really implement "focus-of-attention" forward chaining this way, since we can't insert new facts just after the last fact we selected, only at the beginning (with **asserta**) and end (with **assertz**) of the database. To prevent fact reuse, we can delete facts once pursued. But deleted facts are still important to us (obtaining facts is the whole point of forward chaining) so we'll copy them into **usedfact** facts before we delete them. The revised code:

```
forward :- done.
forward :- fact(F), not(pursuit(F)), assertz(usedfact(F)),
  retract(fact(F)), forward.
```

(Remember from Section 6.1 that **retract** removes a fact from the database.) Then when we're done, all the things we learned plus all the starting facts are in the database as arguments to the **fact** and **usedfact** predicates.

The **pursuit** predicate can cycle through the rules like **one_cycle** did in the hybrid implementation:

```
pursuit(F) :- rule(L,R), rule_pursuit(F,L,R), fail.
```

For **rule_pursuit** we must search through the right side of a rule, deleting anything that matches the fact **F**; we can use the **member** and **delete** predicates of Sections 5.5 and 5.6 respectively. As you may dimly recall, **member** checks whether an item is a member of a list, and **delete** removes all occurrences of an item from a list. We need them both because **delete** always succeeds, and we'd like to fail when a match doesn't occur in the list. So (Figure 7-3):

```
forward :- done.
forward :- fact(F), not(pursuit(F)), assertz(usedfact(F)),
  retract(fact(F)), forward.
pursuit(F) :- rule(L,R), rule_pursuit(F,L,R), fail.
rule_pursuit(F,L,R) :- member(F,R), delete(F,R,Rnew),
  new_rule(L,Rnew).
new_rule(L,[]) :- not(fact(L)), asserta(fact(L)).
new_rule(L,R) :- not(R=[]), asserta(rule(L,R)).
```

The two **new_rule** lines say that when you've deleted everything on the right side of a rule, the left side is a new fact; otherwise just write a new, shorter and simpler rule. And here again are **member** and **delete**:

```
member(X,[X|L]).
member(X,[Y|L]) :- member(X,L).
delete(X,[],[]).
delete(X,[X|L],M) :- delete(X,L,M).
delete(X,[Y|L],[Y|M]) :- not(X=Y), delete(X,L,M).
```

As with the rule-cycle hybrid, you must define **done** appropriately. If you want to make sure that all possible conclusions are reached, use

```
done :- not(fact(X)).
```

which forces forward chaining to continue until there are no more **fact** facts, in which case everything learned (as well as the initial facts) is a **usedfact**. (Note this rule violates our advice in Section 3.6 to avoid **not**s whose variables aren't bound. Here we want to stop if there's any unexamined fact **X** remaining, a kind of existential quantification, so it makes sense. A universally quantified negation is equivalent to an existentially quantified unnegation; see Appendix A.) Alternatively, **done** can be defined to mean that one of a set of "goal" facts have been proved.

The **asserta** in the first **new_rule** rule is important because it forces focus-of-attention handling of new facts. That is, the last fact found will be the next fact pursued, like a stack data structure, because of the recursion. As we pointed out in the last chapter, the focus-of-attention approach often reaches interesting conclusions fast. If we change **asserta** to **assertz**, we get a queue instead of a stack, with new facts pursued only after old facts.

Amazingly, the program works just fine when rules contain variables. This is because the basis step (first line) in the **member** predicate definition can bind variables to succeed, and when it does those variables keep the same bindings through the rest of the rule. If there is more than one matching of a fact to a rule, the program will find each by backtracking to **member**. For instance, the rule

```
rule(a(X,Y),[b(X,Y),b(Y,X),b(X,X)]).
```

can match the fact **b(tom,dick)** to either the first or second predicate expression on its right side, giving two new rules:

```
rule(a(tom,dick),[b(dick,tom),b(tom,tom)]).
rule(a(dick,tom),[b(dick,tom),b(dick,dick)]).
```

When forward chaining with the preceding program is slow, a simple change can often speed things up. That is to delete the old rule when a new rule is formed. This speeds things up because the sum of the number of rules, the number of **fact**s, and the number of **oldfact**s stays constant instead of always increasing. To do this, we need only add a single expression to the **rule_pursuit** rule:

```
rule_pursuit(F,L,R) :- member(F,R), delete(F,R,Rnew),
  retract(rule(L,R)), new_rule(L,Rnew).
```

We can only do this safely for rule-based systems representable as and-or-not lattices, where there are either no variables or only variables that can take a single binding. Otherwise deletion will throw away still-possibly-useful rules, but this may not bother us if the odds are small they're still useful.

Forward chaining with "not"s (*)

As with hybrid chaining, we can avoid rules containing **not**s by substituting "unfact" predicate names representing the opposite of other predicate names. Or we can require that arguments to **not**s never be matchable to anything appearing on the left side of a rule (DeMorgan's Laws can get the rules into this form; see Appendix A). Then we rewrite the top level of the program to handle **not**s after it's done everything else:

```
full_forward :- forward, handle_nots.
handle_nots :- rule(L,R), member(not(X),R), not(usedfact(X)),
  not(fact(X)), delete(not(X),R2), new_rule(L,R2), handle_nots.
handle_nots :- forward.
```

This is not quite the algorithm in Section 6.2, but it's close.

General iteration with "forall" and "doall" (*)

The iteration method of the rule-cycle hybrid and forward chaining programs can be generalized. First, suppose we want to check whether some predicate expression **Q** succeeds for all possible variable values that satisfy some other predicate expression **P**; that is, we want to check *universal quantification* of **Q** with respect to **P**. We can do this by requiring that there be no way for **Q** to fail when **P** has succeeded previously, taking into account any bindings. We can use the built-in **call** predicate of Prolog, which queries a predicate expression given as argument:

```
forall(P,Q) :- not(somefailure(P,Q)).
somefailure(P,Q) :- call(P), not(call(Q)).
```

As an example, assume this database:

```
a(1).
a(2).
b(1).
b(2).
c(1).
c(2).
c(3).
d(1,5).
d(5,1).
```

Here are some example queries and their results:

```
?- forall(a(X),b(X)).
yes
?- forall(b(X),c(X)).
yes
?- forall(c(X),b(X)).
no
?- forall(c(X),d(X,Y)).
no
?- forall(d(X,Y),d(Y,X)).
yes
```

Similarly, we can define a predicate that repeatedly backtracks into predicate expression **P** until **P** fails:

```
doall(P) :- not(alltried(P)).
alltried(P) :- call(P), fail.
```

Assume this database:

```
a(1).
a(2).
a(3).
u(X) :- a(X), write(X).
v(X) :- u(X), Y is X*X, write(Y), nl.
```

Then here are two examples:

```
?- doall(u(X)).
123
yes
?- doall(v(X)).
1
4
9
yes
```

(Remember, **write** prints its argument on the terminal, and **nl** prints a carriage return.)

This **doall** is just what the forward chaining program accomplishes in the **pursuit** predicate. So we can rewrite the first four lines of the forward chaining program as

```
forward :- done.
forward :- fact(F), doall(pursuit(F)), assertz(usedfact(F)),
  retract(fact(F)), forward.
pursuit(F) :- rule(L,R), rule_pursuit(F,L,R).
```

instead of

```
forward :- done.
forward :- fact(F), not(pursuit(F)), assertz(usedfact(F)),
  retract(fact(F)), forward.
pursuit(F) :- rule(L,R), rule_pursuit(F,L,R), fail.
```

And in the rule-cycle hybrid

```
hybrid :- done.
hybrid :- doall(r), hybrid.
```

can be used instead of

```
hybrid :- done.
hybrid :- not(one_cycle), hybrid.
one_cycle :- r, fail.
```

And the code for handling **not**s in forward chaining given in Section 7.11 can be improved to

```
full_forward :- forward, doall(handle_not), forward.
handle_not :- rule(L,R), member(not(X),R), not(usedfact(X)),
  not(fact(X)), delete(not(X),R2), new_rule(L,R2).
```

instead of

```
full_forward :- forward, handle_nots.
handle_nots :- rule(L,R), member(not(X),R), not(usedfact(X)),
  not(fact(X)), delete(not(X),R2), new_rule(L,R2), handle_nots.
handle_nots :- forward.
```

The changes improve program readability.

Input and output of forward chaining (*)

Fact pursuit is only part of what we need for forward chaining. We must also handle input and output differently than with backward chaining. Backward chaining asked the user a question whenever an answer was relevant to some conclusion under study. This can mean that many irrelevant questions are asked before backward chaining hits on the right conclusions to try to prove. Forward chaining, on the other hand, focuses on a set of facts. The facts must get into the database somehow to start things off.

Two approaches are possible. First, give a *questionnaire*, a fixed set of questions to a user presented one at a time, and code answers into facts. Fixed questionnaires are common in the early part presented one at a time of medical diagnosis, when a doctor tries to get a broad picture of the health of a patient before moving on to specifics. Second (especially if most possible facts don't have arguments), give a *menu*, a set of questions presented simultaneously to the user, and ask which questions should be answered **yes**. For diagnosis applications, the menu can contain common symptoms. Menus are good when there are lots of possible facts, few of which are simultaneously relevant to a case.

Both are straightforward to implement in Prolog. Questionnaires can be done by a fixed sequence of calls to the **askif** predicate defined in Section 7.3. The answer to each question will cause the assertion of **asked** and **fact** facts. Menus can be implemented by an **ask_which** predicate | REFERENCE 1|: .FS | REFERENCE 1| It's interesting to compare this program to the forward-chaining program in Section 7.10: this represents more a Lisp style of programming, with recursion through lists and no expectation of backtracking. The forward-chaining program represents more a Prolog style of programming, with frequent backtracking and no lists. .FE

```
ask_which([A,B,C,D,E,F,G,H|L]) :-
screen_ask_which([A,B,C,D,E,F,G,H],[A,B,C,D,E,F,G,H]),
ask_which(L).
ask_which([]).
ask_which(L) :- length(L,N), N<9, N>0, screen_ask_which(L,L).
screen_ask_which([X|L],L2) :- length(L,N), length(L2,N2),
  N3 is N2 - N, write(N3), write(': '), questioncode(X,Q),
  write(Q), write('?'), nl, asserta(asked(X,no)),
screen_ask_which(L,L2).
screen_ask_which([],L2) :-
  write('Give numbers of questions whose answer is yes.'),
  read(AL), create_facts(AL,L2), nl.
create_facts([N|L],L2) :- item(N,L2,I), assertz(fact(I)),
  retract(asked(I,no)), asserta(asked(I,yes)), create_facts(L,L2).
create_facts([N|L],L2) :- not(item(N,L2,I)), create_facts(L,L2).
create_facts([],L2).
item(1,[X|L],X).
item(N,[X|L],I) :- N > 1, N2 is N-1, item(N2,L,I).
member(X,[X|L]).
member(X,[Y|L]) :- member(X,L).
length([],1).
length([X|L],N) :- length(L,N2), N is N2+1.
```

Here's an example use, assuming the **questioncode** definitions for appliance diagnosis given earlier (Section 7.6):

```
?- ask_which
([device_dead,knobs_do_something,lights_out,cord_frayed,
handyperson,familiar_appliance,device_on,just_plugged,in_socket,
smell_smoke,liquids,heats,powerful,has(knobs),has('moving parts'),
has('knobs, switches, or other controls'),
hear(pop),hear('weird noise')]).
1: Does the device refuse to do anything?
2: Does changing the switch positions or turning
the knobs change anything?
3: Do all the lights in the house seem to be off?
4: Does the outer covering of the cord appear to be
coming apart?
5: Are you good at fixing things?
6: Are you familiar with how this appliance works?
7: Is the ON/OFF switch set to ON?
8: Did you just plug the appliance in?
Give numbers of questions whose answer is yes.[3,4,5,7].

1: Is the cord firmly plugged into the socket?
2: Do you smell smoke?
```

```
3: Have any liquids spilled on the appliance just now?
4: Does the appliance heat things?
5: Does the appliance require a lot of power?
6: Does the appliance have knobs?
7: Does the appliance have moving parts?
8: Does the appliance have knobs, switches, or other controls?
Give numbers of questions whose answer is yes.[6,8].

1: Did you hear a pop?
2: Did you hear a weird noise?
Give numbers of questions whose answer is yes.[].
yes
```

To verify that the correct facts were asserted, we can use Prolog's built-in **listing** predicate that takes one argument, a predicate name, and prints out every fact and rule with that name:

```
?- listing(fact).
fact(device_dead).
fact(lights_out).
fact(cord_frayed).
fact(handyperson).
fact(device_on).
fact(has(knobs)).
fact(has('knobs, switches, or other controls')).
yes
```

Fact order is very important to the operation and hence the efficiency of forward chaining (see Figure 6-3). Furthermore, if your definition of the **done** predicate is anything besides **not(fact(X))**, an incorrect fact order may prevent all correct conclusions. So it is a good idea to order question presentation from most important to least important, since facts are entered with **assertz**. To allow some user control, the preceding program asserts choices for each individual menu in user-supplied order. We set the menu size to eight questions in the program, but you can easily change it if you want to improve user control and your terminal screen is tall enough.

For most applications, you shouldn't put every possible fact into menus. You should put the things most common in rules in a first set of menus, and then run forward chaining and see what conclusions you reach. If no "interesting" conclusions are reached (interestingness could mean membership in a particular set), then present further menus to the user (perhaps about things mentioned on the right sides of the shortest remaining rules), assert new facts, and run forward chaining again. You can repeat the cycle as many times as you like. This is a *fact partitioning* trick for rule-based systems, as opposed to the *rule partitioning* trick in Section 7.8. For appliance repair for instance, the first menu could list key classificatory information, such as whether it has a heating element, whether it has mechanical parts, whether it gets electricity from an outlet or from batteries, and what part of the appliance is faulty. After forward chaining on these facts, further menus could obtain a detailed description of the malfunction.

Both menus and questionnaires should not "jump around" too much between topics as backward chaining often does. People are easily confused by rapid shifts in subject, and asking related questions, even irrelevant ones, makes them feel more comfortable. Doctors know this.

Ordering of output (conclusions) presentation is important too for forward chaining. Backward chaining just establishes one conclusion at a time, but forward chaining may establish a whole lot of interesting conclusions about some problem. In fact this is one of its advantages, that it can determine not one but multiple simultaneous problems are causing observed symptoms. If these conclusions are numerous, it's good to sort them by a fixed priority ordering.

Rule form conversions (*)

The rule forms needed for backward chaining, hybrid chaining, and forward chaining are all different. So it would be nice to automatically convert rules from one form to another. That means treating rules as data, something important in many different areas of artificial intelligence, and supported by most dialects of Prolog and Lisp.

Prolog does this with the **clause** predicate in most dialects. This has two arguments representing a left and a right side of a rule. A query on **clause** succeeds if its arguments can match the left and right sides of a database rule. (Remember, facts are rules without left sides.) So for instance

```
?- clause(a,R).
```

binds the variable **R** to the right side of the first rule it can find that has **a** as its left side. The arguments to **clause** can also be predicate expressions containing arguments themselves. So

```
?- clause(p(X),R).
```

will try to find a rule with a **p** predicate of one argument on its left side, and will bind that one argument (if any) to the value of **X**, and **R** to the query representing the right side of that rule | REFERENCE 2|. .FS | REFERENCE 2| Many Prolog dialects require that the predicate name in the first argument be bound, though arguments to that predicate name may have variables. If you have a list of all predicate names, you can iterate over them to do this. .FE

Facts are just rules with no left sides. And **clause** recognizes this, also matching facts in the database as well as rules. Then its first argument is bound to the fact, and the right side is bound to the word **true**. For example, if for the preceding query we have a fact **p(a)** in our database, one answer will be **X=a** with **R** bound to **true**.

To automatically convert a rule to forward chaining form, we can access it with **clause**, then convert the right side of the rule to a list. The second argument of the clause predicate is bound to a query, not a list, so we need a list conversion operation. This is called "univ" and is symbolized in most Prolog dialects by the infix predicate "**=..**", for which the stuff on the left side is a predicate expression and the stuff on the right side is an equivalent list. So we can say

```
?- clause(L,R), R =.. Rlist.
```

and **Rlist** is a list, as we need for forward chaining. Here's the full code for converting all rules in a Prolog database to forward-chaining form:

```
forward_convert :- clause(L,R), not(overhead_predicate(L)),
  R =.. Rlist, new_rule(L,Rlist), fail.
overhead_predicate(new_rule(X,Y)).
overhead_predicate(rule(X,Y)).
overhead_predicate(fact(X)).
new_rule(L,[]) :- not(fact(L)), asserta(fact(L)).
new_rule(L,R) :- not(R=[]), asserta(rule(L,R)).
overhead_predicate(forward_convert).
overhead_predicate(overhead_predicate(X)).
```

If your Prolog dialect doesn't allow **L** to be unbound, you can use:

```
forward_convert(All_predicates) :- member(L,All_predicates),
  forward_convert2(L).
forward_convert2(L) :- clause(L,R), R =.. Rlist, new_rule(L,Rlist), fail.
new_rule(L,[]) :- not(fact(L)), asserta(fact(L)).
new_rule(L,R) :- not(R=[]), asserta(rule(L,R)).
```

and you must query **forward_convert** with a list of all the predicates you want to convert.

But all this won't work in some Prolog dialects that treat commas and semicolons like infix operators. If the preceding doesn't work for you, try:

```
forward_convert(Preds) :- member(Pred,Preds), forward_convert2(Pred).
forward_convert2(Pred) :- clause(Pred,R), remove_commas(R,R2),
  new_rule(Pred,R2), fail.
remove_commas(true,[]).
remove_commas(S,[Y|L]) :- S=..[Comma,Y,Z], remove_commas(Z,L).
remove_commas(X,[X]) :- not(X=true), not(X=..[Comma,Y,Z]).
member(X,[X|L]).
member(X,[Y|L]) :- member(X,L).
new_rule(L,[]) :- not(fact(L)), asserta(fact(L)).
new_rule(L,R) :- not(R=[]), asserta(rule(L,R)).
```

We can apply the same approach to the hybrid control structure. We convert the right side of each backward-chaining rule to a list, insert a **not** expression at the front of this list, insert an **asserta** expression at the end of the list, and replace the original left side with the single symbol **r**. We then can use the **=..** operation in reverse (the left side unbound and the right side bound), converting the list of right-side expressions to a query. We enter this new rule into the Prolog database using **asserta** with this new rule as argument. Here's the full code:

```
hybrid_convert :- clause(L,R), R =.. Rlist,
  add_last(asserta(L),Rlist,Rlist2),
  R2 =.. [not(L)|Rlist2], asserta(r :- R2), fail.
add_last(X,[],[X]).
add_last(X,[Y|L],[Y|L2]) :- add_last(X,L,L2).
```

If your Prolog dialect doesn't allow **L** to be unbound, you can use:

```
hybrid_convert(All_predicates) :- member(L,All_predicates),
  hybrid_convert2(L).
hybrid_convert2(L) :- clause(L,R), R =.. Rlist,
  add_last(asserta(L),Rlist,Rlist2),
  R2 =.. [not(L)|Rlist2], asserta(r :- R2), fail.
add_last(X,[],[X]).
add_last(X,[Y|L],[Y|L2]) :- add_last(X,L,L2).
```

and if none of that works in your Prolog dialect, you can try this alternative hybrid-chaining program which doesn't directly convert Prolog rules but interprets them as needed:

```
hybrid(Leftsidelist) :- done.
hybrid(Leftsidelist) :- not(one_cycle(Leftsidelist)), hybrid(Leftsidelist).
one_cycle(Leftsidelist) :- member(P,Leftsidelist), clause(P,R),
remove_commas(R,R2), allfacts(R2), not(P), asserta(P), fail.
member(X,[X|L]).
member(X,[Y|L]) :- member(X,L).
remove_commas(true,[]).
remove_commas(S,[Y|L]) :- S=..[Comma,Y,Z], remove_commas(Z,L).
remove_commas(X,[X]) :- not(X=true), not(X=..[Comma,Y,Z]).
allfacts([]).
allfacts([Term|L]) :- clause(Term,true), allfacts(L).
```

Indexing of predicates (*)

Prolog interpreters automatically index predicate names appearing on the left sides of rules, to help in backward chaining; that is, they keep lists of pointers to all rules with a particular predicate name, to speed finding them. But this won't help our rule-cycle hybrid and forward chaining implementations, for which we must do our own indexing if we want efficiency.

Let's take pure forward chaining as an example. Indexing could mean storing for each predicate name a list of which rule *right* sides it appears in. One way is to generalize the **rule** predicate of Section 7.10 to a **rules** predicate whose first argument is a predicate expression **P** and whose second argument is a list of pairs representing rules containing **P** on their right side. The first item of each pair is a rule left side, and the second item is a rule right side containing the specified predicate name. So the rules

```
a :- b, c.
d :- c, e, f.
```

are written as

```
rules(b,[[a,[b,c]]]).
rules(c,[[a,[b,c]],[d,[c,e,f]]]).
rules(e,[[d,[c,e,f]]]).
rules(f,[[d,[c,e,f]]]).
```

and just changing **pursuit**, the forward chaining program becomes:

```
forward :- done.
forward :- fact(F), doall(pursuit(F)), assertz(usedfact(F)),
retract(fact(F)), forward.
pursuit(F) :- rules(F,Rlist), member([L,R],Rlist), rule_pursuit(F,L,R).
rule_pursuit(F,L,R) :- member(F,R), delete(F,R,Rnew), new_rule(L,Rnew).
new_rule(L,[]) :- not(fact(L)), asserta(fact(L)).
new_rule(L,R) :- not(R=[]), asserta(rule(L,R)).
```

A rule with K expressions on its right side is repeated K times this way, so this indexing buys speed at the expense of space. The speed advantage comes from eliminating the **member** predicate in the earlier implementation; we have cached in advance the results of running **member**.

Clever indexing techniques have been developed in both artificial intelligence and database research, and much further can be done along these lines.

Implementing meta-rules (*)

Meta-rules are rules that treat rules as data, usually by choosing the one to use next (*conflict resolution*). Choosing means treating rules as data. So meta-rule implementation needs special rule formats like those for hybrid and pure-forward chaining, and can exploit **clause** and **=..** for rule-form conversion. We can represent meta-rules as rules with a special **prefer** predicate name on their left sides. The **prefer** will take four arguments: the left side of a rule 1, the right side of rule 1 (as a list), the left side of a rule 2, and the right side of rule 2 (as a list). It should succeed when rule 1 is preferable (to execute next) to rule 2. (Of course, none of your regular rules can have a **prefer** predicate name for this to work.) Here's an example:

```
prefer(L1,R1,L2,R2) :- length(R1,Len1), length(R2,Len2),
Len1 < Len2.
```

This says a rule with shorter right side should be preferred. Predicate **length** was defined in Section 5.5, and computes the number of items in a list:

```
length([],0).
length([X|L],N) :- length(L,N2), N is N2 + 1.
```

Another example:

```
prefer(a(X),R1,a(Y),R2) :- member(b(Z),R1), not(member(b(W),R2)).
```

This says that if two rules both conclude something about one-argument predicate **a**, one is preferred if it mentions one-argument predicate fact for the second) concluding about a one-argument predicate **a**, whether their arguments are constants or variables.

Meta-rules are most useful when they refer, unlike these two examples, to the current state of reasoning rather than the unvarying text of rules. As an example, consider this meta-rule to be used with pure forward chaining:

```
prefer(L1,R1,L2,R2) :- member(b,R1), not(member(b,R2)), fact(b).
```

This says to prefer a rule that mentions predicate expression **b** to one that doesn't, whenever **b** was proved a fact. A more useful meta-rule says to prefer the rule used most recently:

```
prefer(L1,R1,L2,R2) :- used(L1,R1,T1), used(L2,R2,T2), T1 > T2.
```

Here we assume **used** is a fact asserted after each successful rule application, stating the time a rule was used (times can be increasing integers). Such meta-rules permit flexible control adjustable to the processing context, something general-purpose control structures can't do by themselves.

Programmers can write these meta-rules at the same time they write the rules for an application. The set of meta-rules can express a lattice of orderings (or *partial ordering*) of rule preferences, but not necessarily a complete ordering or sorting. So it's possible for neither of two rules to be preferred to the other, in which case we can pick one arbitrarily.

Meta-rules enhance general-purpose control structures, and aren't a control structure by themselves. This means that meta-rule implementation is different for backward, hybrid, and forward chaining. With backward chaining, meta-rules pick a rule or fact (not necessarily the first in database order) to try to match to a predicate expression in a query or rule right side. With hybrid chaining, meta-rules pick a rule from the entire set of rules to run next. With pure forward chaining, meta-rules can both select a fact to try next and select a rule to try to match the fact to. As an example, here's an implementation of meta-rules with hybrid chaining.

```
metahybrid :- done.
metahybrid :- pick_rule(R), call(R), metahybrid.
pick_rule(R) :- clause(r,R), not(better_rule(R)).
better_rule(R) :- clause(r,R2), prefer(r,R2,r,R).
```

This assumes rules are written in the standard hybrid form with **r** on the left side and the two extra things on the right side. Rules are repeatedly executed (using the **call** predicate, which executes a query as if it were typed in) until the **done** condition is satisfied. A rule is chosen to execute only if no other rules are preferred to it according to the meta-rules.

Implementing concurrency (*)

Several Prolog dialects provide for concurrency of rules and expressions in rules. The emphasis is programmer's tools to indicate good places to do concurrency rather than automatic choice. For instance, a special "and" symbol, used in place of a comma, can specify "and"-parallelism on the right side of a rule. These approaches to concurrency are complicated and incompatible with each other, and we won't discuss them here.

Decision lattices: a compilation of a rule-based system (*)

Compilers and compilation are important concepts in computer science. Compilers take a program in an easy-to-read but slow-to-execute form and convert it to a more efficient, easier-to-interpret form. Compilation is often rule-based itself, especially often-complicated *code optimization*. But compilation techniques can also make artificial intelligence programs themselves more efficient. Since rule-based systems are further steps in power beyond traditional higher-level languages like Pascal, Ada, PL/1, and Fortran, secondary compilations are usually done before the primary compilation to machine language. These secondary compilations convert rules to the format of those languages: formats without references to backward, forward, or hybrid chaining, and with no backtracking, no postponing of variable bindings, and no multiway designation of inputs and outputs to predicate expressions. The decision lattice representation of rules introduced in Section 6.8 is one such secondarily-compiled format. It starts at some node in a lattice, and depending on how a user answers a question, it those to some next node in the lattice. Each question represents another branch point. When it gets to a node that is a *leaf* (a node from which it cannot go any further), it retrieves the conclusion associated with that node.

A decision lattice for a set of rules can be created systematically (albeit not algorithmically, since it involves some subjective judgment) from rules without "or"s on their right sides, by the following (similar to "automatic indexing" methods):

> 1. For every top-level or **diagnosis** rule, repeatedly substitute in the definitions for (right sides of) all intermediate predicates on its right side, until no more remain. If there is more than one rule proving an intermediate predicate, make multiple versions of the diagnosis rule, one for each possibility. (This is a useful compilation method even if you don't want a decision lattice; it's called *rule collapsing*.)
>
> 2. Examine the right sides of the new rules. Pick a predicate expression that appears unnegated in some rules and negated in an approximately equal number (the more rules it appears in, the better, and the more even the split, the better). Call this the partitioning predicate expression, and have the first question to the user ask about it. Create branches from the starting node to new nodes, each corresponding to a possible answer to this question. Partition the rules into groups corresponding to the answers, and associate each group with one new node (copies of rules not mentioning the predicate expression should be put into every group). Then remove all occurrences of the expression and its negation from the rules. Now within each rule group, separately apply this step recursively, choosing a predicate that partitions the remaining rules in the group best, and taking its questioning next.

An example will make this clearer. Suppose we have facts, **a**, **b**, **c**, **d**, and **e**, and possible diagnoses (final conclusions) **r**, **s**, **t**,**u**, and **v**. Suppose these are the rules:

```
r :- a, d, not(e).
s :- not(a), not(c), q.
t :- not(a), p.
u :- a, d, e.
u :- a, q.
v :- not(a), not(b), c.
p :- b, c.
p :- not(c), d.
q :- not(d).
```

For the first step (preprocessing), we substitute in the intermediate predicates **p** and **q**. (Intermediate predicates are just anything that occurs on both a left and a right side.)

```
r :- a, d, not(e).
s :- not(a), not(c), not(d).
t :- not(a), b, c.
t :- not(a), not(c), d.
u :- a, d, e.
u :- a, not(d).
v :- not(a), not(b), c.
```

For the second step we first examine right sides of rules to find something mentioned in a lot of rules that partitions them evenly as possible. Expression **a** is the obvious choice, because it is the only expression occurring in every rule. So we partition on **a**, deleting it from the rules, getting two rule sets:

```
r :- d, not(e).  /* Subdatabase for "a" true */
u :- d, e.
u :- not(d).

s :- not(c), not(d).  /* Subdatabase for "not(a)" true */
t :- b, c.
t :- not(c), d.
v :- not(b), c.
```

The first set will be used whenever the fact **a** is true, and the second set will be used whenever the fact **a** is false. In the first group **d** appears in all rules, so it can be the partitioning expression. Likewise, **c** can partition the second group. This gives four rule groups or *subdatabases*:

```
r :- not(e).           /* The "a, d" rule subdatabase */
u :- e.

u.                     /* The "a, not(d)" subdatabase */
```

```
t :- b.                 /* The "not(a), c" subdatabase */
v :- not(b).

s :- not(d).            /* The "not(a), not(c)" subdatabase */
t :- d.
```

Three of the four groups are two-rule, for which one more partitioning gives a unique answer. The final decision lattice created by this analysis is shown in Figure 7-4.

Decision lattice compilations of rule-base systems can be easily written in any computer language, including Prolog. Give code names to every node in the decision lattice, including the starting and ending nodes. Then define a **successor** predicate of two arguments that gives conditions for one node to be followed by another node. For instance, for our previous example:

```
successor(n1,n2) :- askif(a).
successor(n1,n3) :- askifnot(a).
successor(n2,n4) :- askif(d).
successor(n4,u) :- askif(e).
successor(n4,r) :- askifnot(e).
successor(n2,u) :- askifnot(d).
successor(n3,n5) :- askif(c).
successor(n5,t) :- askif(b).
successor(n5,v) :- askifnot(b).
successor(n3,n6) :- askifnot(c).
successor(n6,t) :- askif(d).
successor(n6,s) :- askifnot(d).
```

Then we query a new **diagnosis** predicate defined:

```
diagnosis(Node,Node) :- not(successor(Node,X)).
diagnosis(D,Start) :- successor(Start,X), diagnosis(D,X).
```

For the preceding example we also need a way to query the facts:

```
questioncode(X,X) :- member(X,[a,b,c,d]), write('Is this correct: ').
member(X,[X|L]).
member(X,[Y|L]) :- member(X,L).
```

And here's what happens:

```
?- diagnosis(X,n1).
Is this correct: a? no.
Is this correct: c? yes.
Is this correct: b? no.
```

```
X=v;
no
```

This program is a simple example of "search" programs, which we'll study in much more detail in Chapters 9, 10, and 11.

Summary of the code described in the chapter (*)

Warning: in using the following, make sure your own code does not redefine or even duplicate the definitions of any predicates used, or you can be in serious trouble.

The following rules assist question-asking for backward, forward, and hybrid chaining. The **ask** predicate asks the user a question, providing extra explanation if the user types a question mark, and returns the answer (after caching it). The **questioncode** and **explain** predicates must be provided by the programmer; the first decodes questions, and the second provides additional explanation for particular questions when the user has trouble understanding. The **askif** predicate handles yes/no questions; it succeeds if the user answers positively, fails if the user answers negatively, and requests another answer if the user answers anything else. Warning: do **abolish(asked,2))** to erase memory before a new situation (problem), if you want to solve more than one situation.

```
/* Tools for questioning the user */
askif(Qcode) :- ask(Qcode,A), positive_answer(Qcode,A).
askifnot(Qcode) :- not(askif(Qcode)).


positive_answer(Qcode,A) :- affirmative(A).
positive_answer(Qcode,A) :- not(negative(A)),
  not(affirmative(A)), write('Please answer yes or no.'),
  read(A2), retract(asked(Qcode,A)),
  asserta(asked(Qcode,A2)), affirmative(A2).


ask(Qcode,A) :- asked(Qcode,A).
ask(Qcode,A) :- not(asked(Qcode,A)), questioncode(Qcode,Q),
  write(Q), write('? '), read(A2), ask2(Q,Qcode,A2,A).


ask2(Q,Qcode,'?',A) :- explain(Qcode), ask(Qcode,A).
ask2(Q,Qcode,A,A) :- not(A='?'), asserta(asked(Qcode,A)).

affirmative(yes).
affirmative(y).
affirmative(ye).
affirmative(right).
affirmative(ok).
affirmative(uhhuh).



negative(no).
negative(n).
negative(not).
negative(never).
negative(impossible).
negative(haha).
```

To do rule-cycle hybrid chaining, write the rules with **r** on their left sides, and an **asserta** of the original left side on the right end of the right side. No rules can contain **not**. Then query **hybrid**, using this program:

```
/* Problem-independent rule-cycle hybrid chaining */
hybrid :- done.
hybrid :- doall(r), hybrid.



doall(P) :- not(alltried(P)).



alltried(P) :- call(P), fail.
```

To do pure forward chaining, write the rules as facts with predicate name **rule**, for which the first argument is a left side and the second argument is the corresponding right side. No rules can contain **not**. Then query **forward**, defined this way:

```
/* Problem-independent forward chaining */
forward :- done.
forward :- fact(F), doall(pursuit(F)), assertz(usedfact(F)),
 retract(fact(F)), forward.



pursuit(F) :- rule(L,R), rule_pursuit(F,L,R).



rule_pursuit(F,L,R) :- member(F,R), delete(F,R,Rnew),
  new_rule(L,Rnew).



new_rule(L,[]) :- not(fact(L)), asserta(fact(L)).
new_rule(L,R) :- not(R=[]), asserta(rule(L,R)).



doall(P) :- not(alltried(P)).
```

```
alltried(P) :- call(P), fail.


member(X,[X|L]).
member(X,[Y|L]) :- member(X,L).


delete(X,[],[]).
delete(X,[X|L],M) :- delete(X,L,M).
delete(X,[Y|L],[Y|M]) :- not(X=Y), delete(X,L,M).
```

When this program stops, the learned facts are left as assertions of **fact** and **usedfact** predicates. If you want to make sure that all possible conclusions are reached, use this definition of **done**:

```
done :- not(fact(X)).
```

If you want to speed up forward chaining, and your rule-based system can be represented as an and-or-not lattice, you can rewrite the preceding **rule_pursuit** rule as:

```
rule_pursuit(F,L,R) :- member(F,R), delete(F,R,Rnew), retract(rule(L,R)),
  new_rule(L,Rnew).
```

If you want to do forward chaining with predicate expressions having **not**s, first make sure the **not**s all refer to fact predicates. Then execute **full_forward**:

```
full_forward :- forward, doall(handle_not), forward.


handle_not :- rule(L,R), member(not(X),R), not(usedfact(X)),
  not(fact(X)), delete(not(X),R2), new_rule(L,R2).
```

Meta-rules can enhance other control structures. They can be written as rules with predicate name **prefer**, of four arguments (the left and right sides of two rules, respectively) that give conditions under which the first rule should be executed before the second rule. For meta-rules with hybrid chaining, execute the query **metahybrid** with this alternative code:

```
/* Problem-independent rule-cycle hybrid chaining using meta-rules */
metahybrid :- done.
metahybrid :- pick_rule(R), call(R), metahybrid.


pick_rule(R) :- clause(r,R), not(better_rule(R)).
```

```
better_rule(R) :- clause(r,R2), prefer(r,R2,r,R).
```

To implement menus as a way of getting facts to do forward chaining, execute **ask_which** with argument the list of question codes for the facts you want to check.

```
/* Menu generation for forward chaining */
ask_which([A,B,C,D,E,F,G,H|L]) :-
  screen_ask_which([A,B,C,D,E,F,G,H],[A,B,C,D,E,F,G,H]),
  ask_which(L).
ask_which([]).
ask_which(L) :- length(L,N), N<9, N>0, screen_ask_which(L,L).



screen_ask_which([X|L],L2) :- length(L,N), length(L2,N2),
  N3 is N2 - N, write(N3), write(': '), questioncode(X,Q),
  write(Q), write('?'), nl, asserta(asked(X,no)),
  screen_ask_which(L,L2).
screen_ask_which([],L2) :-
  write('Give numbers of questions whose answer is yes.'),
  read(AL), create_facts(AL,L2), nl.



create_facts([N|L],L2) :- item(N,L2,I), assertz(fact(I)),
  retract(asked(I,no)), asserta(asked(I,yes)), create_facts(L,L2).
create_facts([N|L],L2) :- not(item(N,L2,I)), create_facts(L,L2).
create_facts([],L2).



item(1,[X|L],X).
item(N,[X|L],I) :- N > 1, N2 is N-1, item(N2,L,I).



member(X,[X|L]).
member(X,[Y|L]) :- member(X,L).



length([],1).
length([X|L],N) :- length(L,N2), N is N2+1.
```

To implement a decision lattice, draw the lattice and label all the nodes with unique names. Write rules defining the branches with the **successor** predicate, for example

```
successor(n5,n9) :- askif(device_dead).
successor(n5,n15) :- askifnot(device_dead).
```

which says that if you're at node n5, go to node n9 if the user responds positively to the **device_dead** question, otherwise go to node n15. Then to run the decision lattice call **diagnosis(D,<first-node>)**, which needs this definition:

```
diagnosis(D,Start) :- successor(Start,X), diagnosis(D,X).
diagnosis(Node,Node) :- not(successor(Node,X)).
```

Keywords:

diagnosis
character string
input coding
output coding
intermediate predicates
menu

.SH Exercises

7-1. (R,A,P) Improve the appliance-diagnosis program given in Section 7.6 so it can reach the following new diagnoses:

> --the motor has burned out, for appliances that have motors;
> --something is blocking the mechanical operation (like something keeping the motor from turning), for mechanical appliances;
> --some internal wiring is broken (possible if appliance was dropped recently, or some other jarring occurred);

Add new questions and perhaps intermediate predicates to handle these diagnoses. Show your new program working.

7-2. (A,P) Figure out a simple way to prevent repeated printing of the same diagnosis in the backward-chaining appliance program as in Section 7.7. Show your method working on the appliance program.

7-3. (E) (a) Medical records often reference very-high-level and very-low-level concepts only, no intermediate concepts. Is this typical of other expert-system application areas?

(b) Suppose you must defend intermediate concepts in an expert system to your boss or representative of a funding agency. They could claim that intermediate concepts don't add capabilities to an expert system, just make its innards neater. Furthermore, intermediate concepts require extra design effort, and slow down operation by forcing reasoning to proceed by smaller steps. How would you reply in defense of intermediate predicates?

7-4. (P,H,G) Design a rule-based expert system to give debugging advice about Prolog programs. Use backward chaining in the manner of the appliance example, asking questions of the user. There is a lot of room for creativity in the choice of what the program knows about, but the program must help debug Prolog programs in some way. Be sure:

1. Your program is a rule-based expert system:

2. Your program contains at least 20 diagnosis rules (rules drawing a conclusion about what is wrong with a Prolog program), 12 of which have more than one kind of evidence "and"ed on their right side;

3. Your program uses at least three intermediate conclusions, conclusions that are neither equivalent to facts nor correspond to advice to the user;

4. Your program can reach at least seven different conclusions depending on the user's answers to the questions;

5. Three of the conclusions your program reaches appear on the left side of more than one rule (that is, there must be multiple paths to three conclusions);

6. Your program uses at least three kinds of history information, like how long the user has been programming Prolog, or whether parts of the user's program are already debugged;

7. The control structure of your program is not extensively "hard-wired" (for instance, there can't be a lot of branch specifications controlling where to go next);

8. As much as possible, all the variables and predicate names are semantically meaningful (that is, their function should be explained by their names; use numbers in your program only for (possibly) probabilities).

Use of probabilities or certainty factors is not necessary. Think about good ways to present your conclusions to the user.

7-5. (A) (For people who know something about probabilities.) Caching of facts in backward chaining is not always a good idea; it depends on the situation. Suppose we have rules for testing whether **a(X)** is true for some **X**, rules that require R units of time to execute on the average. Suppose to speed things up we cache K values of **X** for which the **a** predicate holds. That is, we place facts for those values in front of the rules that calculate **a(X)** as previously. Then to answer a query on the **a** predicate, we first sequentially search through all these cached facts, and use the rules only if we can't find a match in the cache.

(a) Suppose the probability is P that any one item in the cache matches an incoming query, and suppose the cache items are all different and do not have variables, so if one item matches the query no other item will. Under what approximate conditions will using the cache be preferable to not using the cache? Assume that |K < 0.1 / P|, and assume that each cache access requires one unit of time.

(b) Now assume the items of the cache are not all equally likely to be used. Often a *Zipf's Law* distribution applies, for which the probability of the most common item is P, the probability of the second most common item is P/2, the probability of the third most common item is P/3, and so on. Again assume cache items are mutually exclusive. Again assume that each cache access requires one unit of time. Under what approximate conditions is caching desirable now? Hint: the sum of |1 / I| from |I = 1| to |K| is approximately |log sub 2 (K + 1)|.

7-6. (P) Write a rule-based expert system to recommend cleaning methods for clothing. Obtain rules from a human "expert" on the subject. Use hybrid chaining in implementation for simplicity. Ask the user to give a single category that best describes the material they want to clean, and then ask additional questions as necessary to make its description more precise. Write at least twenty rules reaching at least ten different cleaning methods as conclusions. Use a lot of **a_kind_of** facts to classify the many kinds of clothing. You'll need to worry about rule order, and default rules will help.

7-7. The criteria for when to delete a rule in forward chaining were conservative: they miss other circumstances under which it would be good to delete rules. Define those circumstances, and explain how we could efficiently check for them.

7-8. (H,P) Write a program to diagnose malfunctions of a car using forward chaining in the pure form. Write at least fifty rules and use at least ten intermediate predicates. Use a repair manual as your source of expertise. Concentrate on some particular subsystem of the car to do a better job. Use the menu approach to defining facts. Provide for at least two invocations of forward chaining, and stop when one of a set of **diagnosis** facts is proved.

7-9. (H,P) Write a rule-based expert system to diagnose simple illnesses and medical problems (such as a "primary care" or "family practice" physician might handle). Use a medical reference book as your source of knowledge. Your program should be able to reach 30 different diagnoses, at least seven in more than one way, using at least five intermediate predicates. Run your rules with both backward chaining and forward chaining and compare performance. To do this, use the automatic conversion routines, or use the forward-chaining rule form and write your own backward-chainer. For forward chaining, let the user pick symptoms from a series of menus, and ask questions of the user directly for additional secondary facts. Show your rules working on sample situations.

7-10. (P) Write a rule-based expert system to choose a good way to graphically display data. Suppose as input this program loads a file of **data** facts of one argument, a list representing properties of some object. Suppose the output is a recommendation about whether to use bar graphs, pie graphs, two-dimensional plots, summary tables, etc. for particular positions in the data list. For instance, a recommendation might be to plot all the first items of data lists as an X coordinate against second items as a Y coordinate, and to draw a bar graph showing all the first items against third items. These graphing recommendations will need to pick subsidiary information about ranges to be plotted, binning (what values get grouped together), approximate dimensions of the display, extra lines, unusual values (*outliers*) that have been left out, etc. Generally you can find some graphical display for every pair of data item positions, but only some of these will be interesting and worth recommending. Interestingness can be defined in various ways, but should emphasize the unpredictability of the data: a nearly straight line makes an uninteresting graph. It will be useful to define a number of processing predicates, including a **column** predicate that makes a list of all the data values in position K in the data list, and predicate that applies a standard statistical test (look one up in a statistics book) to see if the values in two such columns are associated or correlated. You may want to consult a book giving recommendations for graphical display.

7-11. (A) The definitions of the **forall** and **doall** predicates both use the built-in Prolog predicate **call**, which takes a predicate expression as argument and queries it. (This is useful because the argument can be a variable bound within a program.) Use **call** to implement the following:

(a) **or(P,Q)** which succeeds whenever querying either **P** or **Q** succeeds

(b) **if(P,Q,R)** which succeeds whenever **P** succeeds then **Q** succeeds, or if **P** fails then **R** succeeds

(c) **case(P,N,L)** which succeeds whenever predicate expression number **N** of list **L** succeeds, where **N** is bound by executing predicate expression **P** (that is **N** must be a variable in **P** that is bound by **P**)

7-12. (H,P) Implement the other hybrid of backward and forward chaining mentioned in Section 6.4, the hybrid that alternates between forward and backward chaining. Show your program working on some sample rules and facts.

7-13. (P) Implement meta-rules for pure forward chaining. Use a **prefer** predicate like the one used for rule-cycle hybrid meta-rules.

7-14. (P) Consider meta-rules with pure forward chaining (the implementation is considered in Exercise 7-13). Assume there are no variables in the rules.

(a) Write a meta-rule to prevent the same conclusion from being reached twice.

(b) Write a meta-rule, and a little additional code to that written for Exercise 7-13, to prevent the same rule from being used twice in a row.

7-15. Convert the appliance-diagnosis program to a decision lattice. Try to minimize the number of questions required to a user.

7-16. (H,P,G) Write a rule-based system that, given the syllables of an English sentence in phonemic representation, figures out which syllables to stress. Such stress rules are relatively straightforward and can be found in many linguistics books. You'll need list-processing techniques from Chapter 5, and you must worry about the order the stress rules are applied.

7-17. (P) Write a program to give simple navigation commands to a mobile robot moving across a battlefield. Assume the battlefield is divided into squares, designated by integer X and Y coordinates ranging from X=0 to X=50 and from Y=0 to Y=50; the robot is not allowed to leave this area. The robot starts at location somewhere on the Y=0 line and it is trying to get to location (50,30). There are impassible bomb craters at (20,10), (10,25), and (20,40), all circles of radius 2. There is an impassible ravine extending along the Y=30 line from X=10 to X=50.

The robot begins moving at time 0. The rule-based system to control it should order one of only four actions:

move one unit north, move one unit south, move one unit east, and move one unit west. Each move takes one unit of time. At time T=15 a bomb crater appears 4 units north of the robot; at T=25 a crater appears 3 units west; at T=30 a crater appears 3 units east; and at T=40 a crater appears 4 units east.

The robot should not be able to "see" craters and ravines until it is just at their edges. But suppose the robot knows the coordinates of the goal location and its current coordinates at all times, so it always knows where it is. Try to specify the robot's actions in general terms, that will work for any configuration of craters and ravines.

Try your program out with the robot starting at various locations along Y=0.

7-18. (E) Like most rule-based expert systems, our appliance diagnosis system doesn't reason about causal chains. In other words, it knows that a certain pattern of symptoms signal an underlying cause, but it doesn't know *why* the cause leads to the symptoms, the exact chains of cause and effect that explain each particular symptom. For instance, it knows that when a device isn't working at all, there might be a short in the cord; but it doesn't know the reason is that a short causes the resistance of a cord to be significantly lowered, causing a lot of electricity to flow into the cord, causing a lot of electricity to flow through a fuse, causing the metal conductor in the fuse to heat up, causing it to melt, causing the metal to flow, causing it to break the electrical connection, causing no electricity to go to the appliance cord, causing the appliance to not work at all. For what kinds of appliances and diagnosis circumstances could this lack of causal-chain reasoning be a problem? (We should use a quite different rule-based system then.) In what sense can the appliance expert system of this chapter be seen as a simplification of a more general kind of expert system?

Go to book index

Representing uncertainty in rule-based systems

Numbers are important in most areas of engineering and science. In artificial intelligence, one use of numbers is to quantify the *degree* to which we are certain about something, so we can rank our deductions. This lets us model the world more realistically. We'll show how these kinds of numbers can be easily added to rule-based systems in Prolog, and suggest ways--conservative, middle-of-the-road, and liberal--to manipulate them.

Probabilities in rules

Rules in rule-based systems have so far been absolute: when things are absolutely true on the right side of a rule, then the thing on the left side is absolutely true. But in many real-world situations, inferences or even facts are to some degree uncertain or probabilistic. This is particularly true when facts in a rule-based system represent evidence, and rules represent hunches (hypotheses) explaining the evidence. Examples are the diagnosis rules for a car-repair expert system; many diagnoses can't be certain unless the car is taken apart, but suggestive evidence can be found from the way the car behaves.

Mathematicians have for a long time used probabilities to model degrees of uncertainty in the world. A probability is the fraction of the time we expect something will be true. Other numbers are used in artificial intelligence to represent uncertainty, but probabilities came first, so we'll prefer them here.

As we mentioned in Section 2.10, we can add probabilities as an last argument to Prolog facts. So to say a battery in a randomly picked car is dead 3% of the time, we can put the fact

```
battery(dead,0.03).
```

in the Prolog database | REFERENCE 1|. .FS | REFERENCE 1| Some Prolog implementations don't allow real (decimal) numbers. If yours doesn't, you can represent probabilities by the integer closest to a million times the probability. So 200,000 would represent a probability of 0.2. You'll also have to modify the formulas given later in this chapter so the math will work right. Use 1,000,000 wherever 1 occurs in formulas, and divide all products by 1,000,000, and multiply all two-number quotients by 1,000,000. Addition and subtraction don't have to be modified.

Also, many Prolog implementations that do handle decimal numbers require digits both before and after the decimal point, so you have to say "0.03" instead of just ".03". We'll do that in this book. .FE We can modify predicate expressions in rules similarly. For instance, if 20% of the time when the car won't start it is true the battery is dead, we could write:

```
battery(dead,0.2) :- ignition(wont_start,1.0).
```

We can write different rules for inference of the same fact from different sources of evidence, each with its own probability. So if 50% of the time when the radio won't play the battery is dead:

```
battery(dead,0.5) :- radio(wont_play,1.0).
```

So if we want to reason about whether the battery is dead, we should gather all relevant rules and facts. Then somehow we must combine the probabilities from facts and successful rules to get a cumulative probability that the battery is dead. This we call the *or-combination* issue with probabilities, since you can think of rules with the same left-side predicate name as an implicit "or".

A second issue is that rules can be for a different reason than facts. Consider the preceding rule for the likelihood the battery is dead when the ignition won't start. Suppose we are not sure the ignition won't start--we've tried it a few times, and the car didn't seem to respond. (It might respond if we waited an hour to try it, which would be true if the engine is flooded.) What now is our probability that the battery is dead? It must be less than 0.2, because the new conditions worsen the implication, but how much less? It seems the 0.2 must be combined with the ignition-dead probability. We will call this the *rule versus evidence probabilities* issue.

A third issue is that rules can have "and"s of several predicate expressions on their right sides, and if each has a probability, we must somehow combine those numbers--what we call the *and-combination* issue. Note this is quite different from the "or-combination", because weak evidence that any "and"ed expression is satisfied means weak evidence that the whole "and" is satisfied (a chain is as strong as its weakest link).

Probabilities often arise in artificial intelligence applications when reasoning backward. For instance, if the battery of a car is dead, the car will not start; and if there is a short in the electrical system of a car, the car will not start. Those things are absolutely certain. But if a car will not start, then we must reason backward to figure the likelihood that the battery is dead; we can't be completely certain because another cause like a short in the electrical system could also explain the failure to start. But reasoning backward from effects to causes has many important applications, so we must do it if we want computers to behave intelligently.

Some rules with probabilities

To show the flexibility of probabilities in rules, here are some examples. Suppose 6 times out of 10 when the car won't start, the battery is dead. Then:

```
battery(dead,0.6) :- ignition(wont_start,1.0).
```

The 0.6 here is a *conditional probability*, a probability something happens supposing something else happens. But we could also treat the right-side expression as something that couldn't possibly be uncertain, something that is either true or false. Then we could write the rule with a probability only on the left side:

```
battery(dead,0.6) :- ignition(wont_start).
```

Now suppose the car won't start, and we measure the battery voltage with a voltmeter. Suppose we're not skilled at using a voltmeter. The following rule would apply:

```
battery(dead,P) :- voltmeter(battery_terminals,abnormal,P).
```

This says that the battery is dead with the same probability that the voltmeter measurement is outside the normal range. Not being skilled, we might not be measuring the voltage properly (the terminals might be reversed, or the range setting on the voltmeter might be too low or too high, causing us to incorrectly read no voltage). So the uncertainty of the voltmeter measurement is reflected in the conclusion. Note if **P** is 1 (if we are completely sure of our voltmeter measurement) then **P** is 1 for the battery being dead too.

Suppose we want to rewrite the preceding rule to ignore very weak evidence for the conclusion; this will help avoid unnecessary computation on insignificant things. We can put in an arithmetic comparison:

```
battery(dead,P) :- voltmeter(battery_terminals,abnormal,P),
  P > 0.1.
```

This says that if the voltmeter reading is outside the normal range with probability **P**, and **P** is more than 0.1, then the battery is dead with that same probability **P**.

Now consider what to do when the evidence on the right side of a rule can be certain or uncertain, but when it is certain it does not imply certainty of the left side. This can happen when the right side is a conclusion itself. For instance:

```
battery(dead,P) :- electrical_problem(P2), P is P2 * 0.5.
```

This says that half the time when the car has an electrical problem, the battery is dead. Mathematically, it takes the probability of an electrical problem and multiplies it by 0.5 to get the probability of the battery being dead.

Finally, here's an example of evidence combination. If there is an electrical problem and battery is old, then we suspect (with maybe a probability of 0.9, because evidence is stronger than for the preceding rule) that the battery is dead. But suppose we're not sure of either contributing factor, the problem being electrical or that the battery being old. We must somehow decrease the 0.9 by the uncertainty of the factors. One simple way (the probabilistic-independence-assumption method) is to take the product of 0.9 and the probabilities of the two factors, like this:

```
battery(dead,P) :- electrical_problem(P2), age(battery,old,P3),
   P is P2 * P3 * 0.9.
```

So if we believe there's an electrical problem with probability 0.7, and we're 80% sure the battery is old, an estimate of the probability that the battery is dead is |0.7 * 0.8 * 0.9 = 0.504|.

All probabilities are estimates. We'll treat our probabilities as rough indications of certainty, mainly important relative to one another. We won't insist that all our probabilities of a certain kind sum to 1, because rarely can we feel we've covered all possibilities. For instance, a infinity of things can go wrong with your car, and it wouldn't be reasonable to compute the probability that your car won't start because mutant rats have eaten the engine. (We won't even insist that probabilities of a certain kind must sum to no more than 1, because it's hard to analyze the amount of overlap among probabilities.)

Combining evidence assuming statistical independence

The last section shows that combining probabilities in rule-based systems is very important. Surprisingly, there is no fully general mathematical approach to combining; the problem can be proved mathematically intractable. But we can give some formulas that hold under particular assumptions, and most artificial-intelligence work follows this route. Alas, people make frequent mistakes with probabilities, as shown by experiments, so it's a bad idea to look for guidance from the way people reason.

The easiest assumption we could make is that the different forms of evidence are probabilistically independent. That is, occurrence of one kind of evidence does not make another kind any more or less likely. This situation often happens when the evidence comes by very different reasoning methods, and can't "interact". For instance, suppose we are writing an expert system to give advice about stock-market investments. We might have two rules:

1. If a quarterly report on economic indicators today says that interest rates will go up this year, then the stock market index will go down tomorrow with probability 0.7.

2. If the stock market index has gone up for three straight days, it will go down tomorrow with probability 0.4.

These two rules reflect quite different phenomena. So the success of the first rule won't make us believe the second is any more likely to succeed on the same day, and vice versa. (Maybe there is a little bit of influence--a stock market going down a lot might indirectly cause interest rates to go up--but the connection is pretty weak.) That's what we mean by statistical independence.

When statistical independence applies, probability theory says the probability of both of two uncertain events occurring is the product of the probabilities of each event occurring individually. In general, if events |A, B, C, ...| are statistically independent, then in mathematical notation

```
p   left [ A   "and"   B "and"   C   "and"   ... right ]
= p ( A ) p ( B ) p ( C ) ...
```

where |p (A)| means "probability of event A", etc. This formula defines *and-combination* of the probabilities of the events with the assumption of probabilistic independence.

As for "or"s (unions) of events instead of "and"s (intersections), consider the *Venn diagram* in Figure 8-1. Let regions represent events, so areas of regions represent probabilities, and the area of region A is |p(A)|. Then the area of the region representing the "or" of A and B is the area of A plus that area of B, minus the area in common between both A and B (we counted the area in common twice, hence we must subtract it out once). So since areas correspond to probabilities:

```
p ( A   "or"   B ) = p ( A ) + p ( B ) - p ( A   "and"   B)
```

That last formula applies to *any* events. But when the independence assumption holds, we can think of the Venn diagrams as being drawn in a special way, so that events A and B correspond not to circles but to rectangular regions that cross at right angles. See the top diagram in Figure 8-2. Here the area of the whole square (representing the *universe* or all possible events) is 1, and the probabilities of A and B are each proportional to a distance along the side of the square. So the area of the upper left subrectangle is the probability |p (A "and" B)|, and the area of a rectangle is its length times its width, or is |p (A) p (B)|. Hence substituting in the preceding equation, we get the formula for "or-combination" of two probabilities with the independence assumption:

```
p ( A   "or"   B ) = p ( A ) + p ( B ) - p ( A ) p ( B )
```

We can generalize this to the "or" of three events:

```
p ( A   "or"   B   "or"   C ) = p ( A ) + p ( B) + p ( C ) - p ( A   "and"   B)
- p ( A   "and"   C ) - p ( B   "and"   C ) + p ( A   "and"   B   "and"   C )



= p ( A ) + p ( B ) + p ( C ) - p ( A ) p ( B ) - p ( A ) p ( C )
- p ( B ) p ( C ) + p ( A ) p ( B ) p ( C )
```

Using mathematics we can prove the general formula for the "or-combination" of a set of probabilities assuming probabilistic independence:

```
p   left [ A   "or"   B   "or"   C   "or"   ... right ] =
1 - [ ( 1 - p ( A ) ) ( 1 - p ( B ) ) ( 1 - p ( C ) ) ... ]
```

Prolog implementation of independence-assumption "and-combination"

We can define a predicate that implements the preceding independence-assumption "and-combination" formula. It will take two arguments: an input list of probabilities, and an output number for the combined probability.

```
indep_andcombine([P],P).
indep_andcombine([P|PL],Ptotal) :- indep_andcombine(PL,P2), Ptotal is P2 * P.
```

We just call this predicate as the last thing on the right side of rules, to combine the "and"ed probabilities in a rule. If we had a rule without probabilities like this:

```
f :- a, b, c.
```

we would turn it into a rule with probabilities like this:

```
f(P) :- a(P1), b(P2), c(P3), indep_andcombine([P1,P2,P3],P).
```

That addresses the third issue discussed in Section 8.1. Interestingly, **indep_andcombine** can also address the second issue discussed in Section 8.1, that of modeling rule strengths. Suppose we have a rule:

```
g(P) :- d(P1), e(P2), indep_andcombine([P1,P2],P).
```

The **indep_andcombine** handles uncertainty of **d** and **e**, but the rule itself may be uncertain, meaning that the conclusion **g** has a probability less than 1 even when P1 and P2 are both 1. We could characterize this rule uncertainty itself with a probability, the probability that the rule succeeds given complete certainty of all terms "and"ed on its right side. If this probability were 0.7 for instance, we could rewrite it:

```
g(P) :- d(P1), e(P2), indep_andcombine([P1,P2,0.7],P).
```

In other words, rule uncertainty can be thought of as a "hidden" "and"ed predicate expression) with an associated probability.

Here's an example. Suppose we have the following rule and facts:

```
f(P) :- a(P1), b, c(P2), indep_andcombine([P1,P2,0.8],P).
a(0.7).
b.
c(0.95).
```

Then for the query

```
?- f(X).
```

P1 will be bound to 0.7, and **P2** to 0.95. Predicate **indep_andcombine** computes |0.7 * 0.95 * 0.8 = 0.537|, and **P** is bound to that; so that's **X**, the total probability of predicate **f**.

For rules that refer only to things absolutely true and false, "and-combination" is unnecessary. The rule-strength probability need only be on the left side, as for instance:

```
f(0.7) :- a, b, c.
```

Prolog implementation of independence-assumption "or-combination"

The remaining issue discussed in Section 8.1 was "or-combination" of probabilities. Independence-assumption "or-combination" can be defined analogously to independence-assumption "and-combination" but using the last formula in Section 8.3:

```
indep_orcombine([P],P).
indep_orcombine([P|PL],Ptotal) :- indep_orcombine(PL,P2),
 Ptotal is 1 - ( (1-P) * (1-P2) ).
```

"Or-combination" is needed when we have multiple evidence for the truth of predicate **f**, as for instance:

```
f(0.5) :- a.
f(0.7) :- b, c.
f(0.8) :- d, not(e).
```

"Or-combination" is more awkward than "and-combination" because we must use a new predicate name to represent the combination | REFERENCE 2|. So for the preceding example we must define something like **f_overall** that represents the cumulative likelihood of **f**. This can use a special built-in predicate called **bagof**, used this way:

```
f_overall(P) :- bagof(P2,f(P2),PL), indep_orcombine(PL,P).
```

We'll explain this **bagof** predicate more formally in Section 10.6. For now we'll note that **bagof** has three arguments: a variable (an input), a predicate expression containing that variable (an input), and a list of all values to which the variable in the expression can successfully be bound (an output). So the **bagof** says to make a list PL of all the P2 such that **f(P2)** succeeds. If the argument to **f** represents the probability of event **f**, the list PL will contain all the probabilities found for the predicate **f** by every possible rules and fact. These can be combined with the **indep_orcombine** | REFERENCE 2|. .FS | REFERENCE 2| An alternative simpler implementation of indep_orcombine is possible in many dialects of Prolog with the "univ" feature (see Section 7.14) that converts from lists to predicate expressions, symbolized by "**=..**", where the left side is an expression and the right side is its component list of symbols. It's used like this: **new_indep_orcombine(F,P) :- Pred =.. [F,P2], bagof(P2,Pred,PL), indep_orcombine(PL,P).** .FE

As an example, suppose we have these rules and facts:

```
g(0.7) :- a.
g(P) :- b(P1), c(P2), indep_andcombine([P1,P2,0.9],P).
g(0.3) :- a, d.
a.
b(0.8).
c(0.9).
d.
```

Then to combine evidence for the **g** predicate we need:

```
total_g(P) :- bagof(P2,g(P2),PL), indep_orcombine(PL,P).
```

and to use it we must query:

```
?- total_g(X).
```

All three **g** rules will succeed with these facts, and **P** in the second **g** rule will be bound to |0.8 * 0.9 * 0.9 = 0.648|. So **PL** in the **total_g** rule will be bound to **[0.7,0.648,0.3]**. Now we must call on **indep_orcombine**. |1 - ((1 - 0.648) * (1 - 0.3)) = 1 - 0.352 * 0.7 = 0.7536|, and |1 - ((1 - 0.7) * (1 - 0.7536)) = 1 - 0.3 * 0.2464 = 0.92608|. Hence the argument **P** to **total_g** will be bound to 0.92608, and hence to **X**, the total probability that **g** is true.

The conservative approach

Independence when combining probabilities is a strong assumption. It does not hold when one event causes another, or when two events are both caused by some other event. For instance with a small appliance, a totally nonfunctioning device suggests an electrical problem, and a frayed cord suggests an electrical problem. This could be represented as:

```
electrical_problem(0.5) :- doesnt_work.
electrical_problem(0.6) :- frayed_cord.
```

If both rules succeed, the independence-assumption probability of an electrical problem is 0.8. But that's too high, because the cord problem could explain the not-working observation: the cord being frayed could cause the device not to work. So the true combined probability should be closer to 0.6, the number in the second rule.

One approach when independence does not hold is to be very conservative, very careful not to overestimate the cumulative probability. This *conservative* approach is sometimes called a *fuzzy set* approach, actually it is more general than what is called "fuzzy set theory", representing a mathematically provable lower bound. Consider a "conservative orcombine". Whatever the total probability for some event with multiple positive evidence, it must be no worse than the probability of the strongest evidence: positive evidence shouldn't ever disconfirm other positive evidence. So we could define a "conservative orcombine" to operate on probability lists in place of **indep_orcombine**, to use whenever independence clearly doesn't hold between the probabilities:

```
con_orcombine(PL,P) :- max(PL,P).
max([P],P).
max([P|PL],P) :- max(PL,P2), P > P2.
max([P|PL],P2) :- max(PL,P2), not(P > P2).
```

The **max** definition is from Section 5.5. The middle diagram of Figure 8-2 is the Venn diagram for the "conservative-or" case.

For a corresponding "conservative andcombine", we could just give "0"--that's always plenty conservative. But if the "and"ed probabilities are all large, we can prove a nonzero value. Consider the two probabilities 0.7 and 0.8 for two events. 30% of the time the first event must be false. That 80% for the second event can't all "fit" into 30%; only part of it can, with another 50% left over. So at least 50% of the time both events must occur; i.e., the minimum probability is 0.5.

In general, the conservative value for |p (A "and" B) | is

```
maxfunction ( p ( A ) + p ( B ) - 1 , 0 )
```

where "maxfunction" is a mathematical function having a value (not a predicate like the previous **max**), and its value is the larger of its two arguments. We will define:

> *The value of maxfunction(X,Y) is the larger of the two numbers X and Y;*
> The value of minfunction(X,Y) is the smaller of the two numbers X and Y.

To generalize the "and" formula to any number of "and"ed expressions, we can define:

```
con_andcombine([P],P).
con_andcombine([P|PL],0) :- con_andcombine(PL,P2), P + P2 < 1.0.
con_andcombine([P|PL],Ptotal) :- con_andcombine(PL,P2),
  P3 is P+P2, not(P3<1.0), Ptotal is P + P2 - 1.0.
```

The bottom diagram in Figure 8-2 is the Venn diagram for the **conservative_and** case. So the diagram for conservative "or" is different from the diagram for conservative "and". In general, you should use the conservative "or" when there are strong positive correlations between evidence, and the conservative "and" when there are strong negative correlations.

To illustrate, let's use the same example of the last section but substitute in **con_orcombine** and **con_andcombine**:

```
total_g(P) :- bagof(P2,g(P2),PL), con_orcombine(PL,P).
g(0.7) :- a.
g(P) :- b(P1), c(P2), con_andcombine([P1,P2,0.9],P).
g(0.3) :- a, d.
a.
b(0.8).
c(0.9).
d.
```

The second **g** rule binds its **P** to |0.8 + (0.9 + 0.9 - 1) - 1 = 0.6|. Then PL is bound to **[0.7,0.6,0.3]**, and **P** in **total_g** is bound to 0.7.

Note we don't have to pick the independence-assumption or conservative approach exclusively in a rule-based system. We can choose one in each situation based on our analysis of appropriateness.

The conservative "or-combination" is particularly useful for handling prior (or *a priori*) probabilities. These are "starting" probabilities for the likelihood of an event on general grounds. They are usually numbers close to 0, and they can be expressed in the Prolog database as *facts* (instead of rules) with probabilities. For instance, checking under the hood of a car for frayed wires is a lot of work, so an expert system for auto diagnosis might first try to diagnose an electrical problem without asking for such an inspection, using an "a priori" probability of 0.01 of any wire being frayed. The independence assumption is a poor idea for combining a priori probabilities because they represent a summary of many sources of evidence.

The liberal approach and others

There's also a *liberal* approach: compute the maximum probability consistent with the evidence. This isn't as useful as the "conservative" approach, but applies for "and"s whenever one piece of evidence implies all the others, and applies for "or"s whenever pieces of evidence are disjoint (i.e., prove the same conclusion is ways that cannot hold simultaneously). Liberal formulas can be derived from the conservative formulas by relating "and" and "or", as with the formula for two variables:

$$p (A \text{ "or" } B) = p (A) + p (B) - p (A \text{ "and" } B)$$

which can also be written

$$p (A \text{ "and" } B) = p (A) + p (B) - p (A \text{ "or" } B)$$

Since $|p (A)|$ and $|p (B)|$ are known, the maximum value of the first formula occurs when $|p (A \text{ "and" } B)|$ has a minimum, and the maximum of the second formula occurs when $|p (A \text{ "or" } B)|$ has a minimum (note the important minus signs). These needed minima are given the conservative formulas. So the liberal bound on two-argument "or"s is

$$p (A) + p (B) - \text{maxfunction} (p (A) + p (B) - 1 , 0)$$

$$= - \text{maxfunction} (-1 , - p (A) - p (B)) = \text{minfunction} (1 , p (A) + p (B))$$

and the liberal bound on "and"s is

```
p ( A ) + p ( B ) - maxfunction ( p ( A ) , p ( B ) )

= - maxfunction ( - p ( B ) , - p ( A ) ) = minfunction ( p ( A ) , p ( B ) )
```

To generalize, the liberal approach for "and-combination" is the minimum of the probabilities:

```
lib_andcombine(PL,P) :- min(PL,P).
min([X],X).
min([X|L],X) :- min(L,X2), X < X2.
min([X|L],X2) :- min(L,X2), not(X < X2).
```

(The **min** is just like the **max** of Sections 5.5 and 8.6.) Similarly, the general "or-combination" is the sum of the probabilities, provided this number is not greater than 1:

```
lib_orcombine(PL,1.0) :- sumup(PL,P), P > 1.0.
lib_orcombine(PL,P) :- sumup(PL,P), not(P > 1.0).
sumup([P],P).
sumup([P|PL],Ptotal) :- sumup(PL,P2), Ptotal is P + P2.
```

The middle diagram in Figure 8-2 shows the liberal "and" case graphically, and the bottom diagram in Figure 8-2 illustrates the liberal "or". That is, they're the situations for the conservative "and" and "or" reversed. In general, you should use the liberal "and" when there are strong positive correlations between evidence, and the liberal "or" when there are strong negative correlations--just the opposite of the advice for the conservative formulas.

Consider the same example we have used before, but with **lib_orcombine** and **lib_andcombine**:

```
total_g(P) :- bagof(P2,g(P2),PL), lib_orcombine(PL,P).
g(0.7) :- a.
g(P) :- b(P1), c(P2), lib_andcombine([P1,P2,0.9],P).
g(0.3) :- a, d.
a.
b(0.8).
c(0.9).
d.
```

Now **P** in the second **g** rule will be bound to 0.8, the minimum of **[0.8,0.9,0.9]**. Then **PL** is bound to **[0.7,0.8,0.3]**, and **P** in **total_g** is bound to 1.0.

Again, we can use the liberal approach wherever we like in a rule-based system, and the conservative and independence-assumption approaches elsewhere in the same system too. The formulas are summarized in Figure 8-3. (All the formulas are associative, so we can get the n-item formula from the two-item formula.) If we aren't sure any is best, we can take a weighted average. Or we could invent our own formula. But we must be careful, because not all formulas make sense. Reasonable criteria are (1) smoothness, that the formula never makes abrupt jumps in value as input probabilities smoothly vary; (2) consistency, that it never gives values outside the range between the conservative and liberal values; (3) commutativity, that the order of binary combination doesn't matter; and (4) associativity, that the formula gives the same result no matter how the expressions are grouped (for example, combining |p sub 1| with the combination of |p sub 2| and |p sub 3| must be the same as combining the combination of |p sub 1| and |p sub 2| with |p sub 3|).

Negation and probabilities

Conditions in rules that something *not* be true cause problems for probabilities. For instance:

```
a :- b, not(c).
```

If a, b, and c are all uncertain and the rule itself has a certainty of 0.7, it won't do to just add extra arguments like this:

```
a(P) :- b(P1), not(c(P2)), indep_andcombine([P1,P2,0.7],P).
```

because if there's any evidence for c, no matter how weak, the rule will fail. We want instead for weak evidence for c to *decrease* the probability of a in a small way. The way to handle this is to note the probability of something being false is one minus the probability of it being true. So instead:

```
a(P) :- b(P1), c(P2), inverse(P2,NegP2), indep_andcombine([P1,NegP2,0.7],P).
```

where **inverse** is defined as:

```
inverse(X,IX) :- IX is 1 - X.
```

So we shouldn't use **not**s when probabilities are involved, but use this predicate **inverse** on the resulting probabilities. (But you must still be careful to remember that **p(0.0)** won't match **not(p)**.)

Some artificial-intelligence systems don't follow this approach, however. They try to be more general by reasoning separately about events and their negations. They collect evidence for an event, and combine it with probability combination rules, but they also collect evidence against an event and combine it separately. Then they combine these two cumulative probabilities somehow to get an overall likelihood measure. A simple way used in many expert systems is to take the difference of the probability for something and the probability against something. This number ranges from 1.0 (complete certainty of truth) through 0.0 (complete indecision) to -1.0 (complete certainty of falsity).

An example: fixing televisions

Now we'll give an example of a simple rule-based expert system using probabilities, for diagnosis of malfunctioning equipment. Unfortunately, most expert systems (like most artificial intelligence programs) must be big to do anything worthwhile; otherwise, human beings could do the job fine without them. So to avoid burdening you with an example ten pages long, we must pick something simple and not very useful. So here's an example of the few things wrong with a television set that you can fix yourself (television sets use high voltages so most malfunctions should be treated by trained service personnel.)

When a television set is working improperly, one of two things may be improperly adjusted: the controls (knobs and switches) or the receiver (antenna or cable). So let's write an expert system to estimate probabilities of those two things. We'll assume these probabilities need not be very accurate, but their relative sizes provide a rough guide to where things are wrong.

Consider why the knobs might be adjusted wrong on a television. If it is old and requires frequent adjustment, that could be a reason. Similarly, if kids use your set, and they play with the knobs, that could be a reason too, as well as anything strange you've done lately that required adjustment of the knobs (like taking a photograph of the television picture, requiring that the brightness be turned up very high). Let's write the rules. If a set requires frequent readjusting, then it's quite reasonable that the set is maladjusted today--let's say 50% sure:

```
maladjusted(0.5) :- askif(frequent_adjustments_needed).
```

(The **askif** predicate was defined in Section 7.3; it types out a question for the user, and checks if the response is positive or negative.) For recent unusual usage of the television, it matters how you define "unusual". But let's say 50% to be reasonable, so the rule is:

```
maladjusted(0.5) :- askif(recent_unusual_usage).
```

The rule for children must insist both that children hang around your house and that they would be inclined to mess around with the knobs on your television. That's the "and" of two conditions, so we need an "andcombine". When both conditions hold, it's quite likely the set is maladjusted, so we can give this rule a rule strength of 0.9. So the rule is:

```
maladjusted(P) :- askif(children_present(P1)),
  askif(children_twiddle_knobs(P2)), andcombine([P1,P2,0.9],P).
```

Then to get the cumulative probability the set was recently adjusted, we need an orcombine:

```
recently_adjusted(P) :- bagof(P2,maladjusted(P2),PL), orcombine(PL,P).
```

This last predicate just summarizes predisposing evidence for a set maladjustment, but it doesn't incorporate the best evidence at all, observations of the television set. In other words, two major factors, past and present, must be combined. This could be either an "andcombine" or an "orcombine", but "andcombine" seems preferable because neither factor here implies strongly the diagnosis; it's only when both occur together that evidence is strong. That suggests:

```
diagnosis('knobs on set require adjustment',P) :-
  recently_adjusted(P2), askif(operation(abnormal)),
  andcombine([P2,0.8],P).
```

(Remember, single quotation marks (') indicate character strings in Prolog; everything between two single quotation marks, including spaces, is treated as a unit.)

If there is some uncertainty about whether the television's behavior is normal, we could include a probability as a second argument to the **operation** predicate, combining it in the "andcombine" too. Or we could characterize the operation by many different words. For instance:

```
diagnosis('knobs on set require adjustment',P) :-
  recently_adjusted(P2), askif(operation(mediocre)),
  andcombine([P2,0.5],P).
```

So we have a three-level expert system: an "and" of two expressions, one of which is an "or" of three expressions, one of which in turn is an "and" of two expressions. It's true we could simplify this into two levels by the laws of logic (see Appendix A), rewriting everything in disjunctive normal form or conjunctive normal form (see Appendix A), but this isn't a good idea with rule-based systems. For one thing, extra levels let you group related concepts together to make the rule-based system easier to understand; in a normal form, widely different predicates can be thrown together. Grouping related terms together also enables easier determination of probability values and easier choice of probability combination methods; it's difficult to pick a good combination method for twenty terms, since some of the "and"ed expressions must be considerably more related than others. Many-level rule-based systems also allow easier design and debugging, because they give lots of places to put checkpoints and tracing facilities.

Now let's turn to the other kind of television diagnosis we can make, that the antenna or cable connection to the television set is faulty. For this conclusion we will use the same **diagnosis**, and **operation** predicates as before. But we'll need a new predicate to summarize contributing factors to the faultiness of the antenna or cable connection, **overall_source_problems**:

```
source_problems(0.8) :- askif(television(new)).
source_problems(0.95) :- askif(antenna(new)).
source_problems(0.95) :- askif(cable(new)).
source_problems(0.3) :- askif(recent_furniture_rearrangement).
overall_source_problems(P) :- bagof(P2,source_problems(P2),PL),
  orcombine(PL,P).
```

Let's assume that no one has both an antenna and a cable connection (or if they do, only one of them is operating). We'll use a **source_type** predicate to indicate whether the set up is an antenna or a cable. Then we have two rules for the two diagnoses:

```
diagnosis('antenna connection is faulty',P) :-
  askif(source_type(antenna)), askif(operation(abnormal)),
  overall_source_problems(P).
diagnosis('cable connection is faulty',P) :-
  askif(source_type(cable)), askif(operation(abnormal)),
  overall_source_problems(P).
```

So that's our simple expert system. To use it, we query

```
?- diagnosis(D,P).
```

And each answer that the Prolog interpreter finds to the query will bind **D** to a string representing a diagnosis, and bind **P** to the corresponding probability. To find all diagnoses with nonzero probabilities, we can repeatedly type semicolons.

Graphical representation of probabilities in rule-based systems

The logic-gate representation of an and-or-not lattice (see Section 6.10) is a useful graphical notation for simple rule-based systems. It can be used for rules with probabilities too. Associate every "andcombine" and "orcombine" with a logic gate in the representation. Indicate rule strengths next to their corresponding logic gates. For rules with only one expression on their right side, use special triangle gates (*attenuators*) with one input and one output. Then each line has an associated probability, computed by proceeding from the inputs through the network, applying the proper formula at each gate. Figure 8-4 shows the and-or-not lattice for the example of the last section.

Getting probabilities from statistics

There's a more fundamental problem with probabilities than combining them, however: getting them in the first place. If probabilities are markedly incorrect, reasoning based on them can't be trusted. But getting good probabilities is often the hardest problem in building a rule-based system. Even when programmers can easily decide what the predicates should be, what things rules should cover, and how rules should be structured, they often have trouble estimating probabilities because it's hard to tell when an estimate is wrong. Two approaches are used: getting probabilities from statistics on data, and getting probabilities from human "experts".

Since people reason poorly about uncertainty, the first approach seems preferable. Often we have a lot of routinely-collected data about the phenomena in a rule-based system, as when our rule-based system further automates human capabilities already partly automated. We can approximate needed probabilities by frequency ratios in the data. For instance, we can approximate rule-strength probabilities by the ratio of the number of times the left side of a rule was satisfied to the number of times the right side was satisfied.

As an example, consider the repair of airplanes. They are expensive, so an organization that owns many of them must keep detailed repair and maintenance records to ensure quality work, to better allocate resources, and to track down trends in malfunctions. Most organizations today computerize these, recording observed malfunctions, their eventually inferred causes, and what was done to fix them--just what we need to assign probabilities to an expert system for diagnosis. For instance, we can count how many times the radar system failed, and how many of those times the widget component was faulty, and take the ratio of the two counts to fill in **<strength>** in the rule:

```
faulty(widget,<strength>) :- failed(radar).
```

However, approximating a real number (a probability) by the ratio of two integers can be hard, and the approximation is often poor when the integers are small. Suppose some event happens with probability 0.001, and we have data for 2000 occurrences. On the average we'll expect two events in those 2000, but the number could be 1, 0, 3, or 4 too, since the event is so rare. According to probability theory, random sets of size N, drawn from an infinite population with fraction F of its members possessing some property, will tend to show the same fraction F with standard error (standard deviation of this fraction) approximately

```
sqrt { F ( 1 - F ) / N }
```

(using the binomial-distribution approximation). This says how good a probability estimate is; the larger this number, the worse the estimate. As a rule of thumb, if the preceding is the same or larger than F, the F fraction should not be trusted.

As an example, suppose something happens 7 times out of 20 possible times. Then |N = 20|, |F = 0.35|, and the standard error by the formula is 0.105. This is significantly less than 0.35, so the probability estimate 0.35 looks OK.

Probabilities derived from others

If a probability is important to our rule-based system, yet the associated standard error of approximation from data is large, we may be able to better estimate the probability from other data. One way is Bayes's Rule. Letting |p (A given B)| represent the probability of event A happening when event B also happens, we can say:

```
p ( A   given   B ) = { p ( A   "and"   B ) }   /   {  p ( B ) }
```

But switching A and B in that equation:

```
p ( B   given   A ) = { p ( A   "and"   B ) }   /   {  p ( A ) }
```

We can solve the second equation for |p (A "and" B)|, and substitute in the first equation, obtaining the usual form of Bayes's Rule:

```
p ( A   given   B ) = { p ( B   given   A ) * p ( A ) }   /   { p ( B ) }
```

This is a useful when we have a rule

```
a(P) :- b(P2), andcombine([P2,<strength>],P).
```

and we want to know what number to put for **<strength>**, the probability that **a** is true given that **b** is true. If we have enough data to compute the reverse--the probability that **b** is true given that **a** is true--then Bayes's Rule can use that number, together with estimates of the overall probabilities of **a** and **b**, to give what we need. This is an especially good idea when **a** *causes* **b**, because then the reverse probability that **b** is true given that **a** is true must be 1.0. For instance, a car absolutely will not start when its battery is dead, so we can approximate the probability that the battery is dead when the car won't start by the ratio of: the overall probability the battery is dead over the overall probability the car won't start.

There are extensions of Bayes's Rule, for instance:

```
p ( A   given   ( B   "and"   C ) ) = { p ( ( B   "and"   C )   given   A ) * p ( A )
}
  /    { p ( B   "and"   C ) }
```

Another trick to get rule probabilities is to use the independence assumption in a special way. Suppose we have:

```
a(P) :- b(P1), c(P2), andcombine([P1,P2,<strength>],P).
```

Now there may be few situations in the data in which both **b** and **c** were true. But if there were many situations in which one of **b** and **c** was true, then we could estimate the probabilities of **a** given **b**, and **a** given **c**, and take the "or-combination" of these two numbers (yes, "or" not "and"; think about it) as an estimate of the rule strength needed. That is, we can precompile an "orcombine".

Subjective probabilities

Even with these tricks we may not have enough data (or perhaps good enough data) to approximate probabilities very well. Then we must guess probabilities ourselves, or preferably, ask a human expert in the task or domain of the rule-based system. This isn't always easy: experts may be hard to find, or their time may be expensive, or they may not understand or feel comfortable with a rule formulation of their knowledge. But there may be no other choice.

As we've said, humans make many mistakes in probability estimation, as demonstrated by psychological experiments. One simple way to make things easier is to let people quantify uncertainty on a different numeric scale than 0.0 to 1.0. For instance, take *degrees of certainty* on a scale 0 to 100, and divide by 100 to get the probability. Better yet, do a nonlinear transformation of the probability scale, for instance with *odds* defined as |p / (1 - p)|. Odds range from 0 to positive infinity, so a probability of 0.9 is odds of 9, a probability of 0.5 is odds of 1, and a probability of 0.1 is odds of 0.111. The logarithm of the odds is also useful; it "flattens out" the curve more, and ranges from minus infinity to plus infinity.

Something that also helps people is speaking of uncertainty nonnumerically. For instance, let them use the terms "certain", "almost certain", "likely", "suggestive", "possible", "not likely", and "impossible". Each term may map to a probability--say 1.00 for "certain", 0.99 for "almost certain", 0.8 for "likely", 0.5 for "suggestive", 0.2 for "possible", 0.05 for "not likely", and 0.0 for "impossible". If this isn't possible, perhaps different probabilities can be given for different contexts, so a "possible car problem" would be a 0.2 probability, but a "possible nuclear accident" would be 0.001 probability.

Maximum-entropy probabilities (*)

Bayes's Rule extends the utility of both statistics and subjective probability estimates. We can generalize this idea, to accept arbitrary probabilities from the programmer--prior probabilities, conditional probabilities, and joint probabilities--and make "reasonable guess" estimates of others, using some mathematics.

It can be shown mathematically that best guesses (based on certain postulates for guess desirability) are those that maximize entropy, or minimize the *information content*, of probability assignments. This is a mathematical optimization problem, in which we want to maximize

```
sum from {i=1} to m left ( - p ( A sub i ) log ( p ( A sub i ) ) right )
```

for some mutually exclusive set of probabilities |p (A sub i)| that sum to 1, subject to given equality constraints in the form of probabilities already known. Optimization problems like this can be attacked by many methods from operations research, and computer packages are available. But they can take time since they usually iterate.

Sometimes we don't need to iterate to find maximum-entropy probabilities, but we can use algebraic manipulations to get formulas. Here's an example for those of you that know some calculus. Suppose we know the probabilities of two events A and B, |p (A)| and |p (B)|. Suppose we want to find the maximum-entropy probability for |x = p (A "and" B)| (i.e., we want to do "and-combination" in a maximum-entropy way). Then there are four mutually exclusive probabilities involved: |p (A "and" B) , p (A bar "and" B) , p (A and B bar) , p (A bar "and" B bar)|, where we use |A bar| to represent the exact opposite of A, so |p (A bar) = 1 - p (A)|. Then

```
p ( A   "and"   B ) = x   ,    p ( A bar   "and"   B ) = p ( B ) - x   ,
p ( A   "and"   B bar ) = p ( A ) - x   ,

p ( A bar   "and"   B bar ) = 1 - p ( A ) - p ( B ) + x
```

And the preceding summation formula for the entropy is

```
- x log ( x ) - ( p ( B ) - x ) log ( p ( B ) - x ) -
( p ( A ) - x ) log ( p ( A ) - x )

 - ( 1 - p ( A ) - p ( B ) + x ) log ( 1 - p ( A ) - p ( B ) + x )
```

To find the maximum of this, we take the derivative with respect to x, and set this to zero. Noting that the derivative of |y log (y)| with respect to x is |(1 + log (y)) dy / dx |, we get:

```
- log ( x ) + log ( p ( B ) - x ) + log ( p ( A ) - x )
- log ( 1 - p ( A ) - p ( B ) + x ) = 0



log [ x ( 1 - p ( A ) - p ( B ) + x ) / ( p ( A ) - x ) ( p ( B ) - x ) ] = 0



x sup 2 - ( p ( A ) + p ( B ) ) x + p ( A ) p ( B ) =
x sup 2 + ( 1 - p ( A ) - p ( B ) ) x



x = p ( A ) p ( B)
```

This is just the independence-assumption formula. So the formula we justified intuitively in Section 8.3 has a deeper justification. Formulas for more complicated situations can also be derived with the method.

Consistency (*)

Another problem with subjective probabilities (but also to a lesser extent with data-derived probabilities) is that they can be inconsistent (logically impossible) in a nonobvious way. This often happens when both a priori (unconditional) and conditional probabilities are specified by people. We should therefore run checks on user-given probabilities before entering them into a rule-based system.

As an example, note from the definition of conditional probability that

$$0 <= p (A \text{ given } B) p (B) = p (B \text{ given } A) p (A) <= 1$$

Hence

$$p (A) >= p (A \text{ given } B) p (B)$$

$$p (B \text{ given } A) >= p (A \text{ given } B) p (B)$$

$$p (B) >= p (B \text{ given } A) p (A)$$

$$p (A \text{ given } B) >= p (B \text{ given } A) p (A)$$

so we can catch some inconsistencies from inequalities.

Keywords:

probability
uncertainty
or-combination
and-combination
rule probability
conclusion probability
independence assumption
conservative assumption
liberal assumption
Bayes's rule
scale transformations
maximum-entropy estimates

.SH Exercises

8-1. Assume:

1. The battery is defective with certainty 0.5 when a car won't start.
2. The battery is defective with certainty 0.8 when the radio is functioning and the radio won't play.
3. You are not sure if your radio is functioning--the probability is 0.9 that it is functioning.
4. This morning your car won't start and the radio won't play.

What is the cumulative probability that your battery is defective this morning? Combine evidence assuming independence of probabilities.

8-2. Consider these rules (the arguments are all probabilities):

```
a(P) :- b(P2), P is P2 * 0.6.
a(P) :- c(P).
```

Suppose **b** is known to be absolutely certain, and **c** is 80 percent certain.

(a) What is the cumulative probability of **a** using the independence assumption?

(b) What is the cumulative probability of **a** using the conservative assumption?

(c) What is the cumulative probability of **a** using the liberal assumption?

8-3. (R,A) Suppose we want to fill in the **<prob1>** and **<prob2>** probability values in the following two rules that infer a flat tire on a car:

```
flat_tire(<prob1>) :- car_steers_strangely.
flat_tire(<prob2>) :- just_ran_over_something.
```

(a) Suppose we have statistics that say:

> -- In 200 situations in which the car steered strangely the tire was flat, out of 500 situations in which the car steered strangely;
>
> -- In 800 situations in which you just ran over something the tire was discovered to be flat, out of 1600 situations in which you just ran over something;
>
> -- A flat tire was observed in 1200 situations total.

Estimate **<prob1>** and **<prob2>** for these statistics.

(b) Suppose we also know that 70 times in which both the car steered strangely and you just ran over something the tire was then found to be flat, out of 101 times in which both those two things were observed. Which probability combination method (or-combination) for the preceding two rules is best confirmed here: conservative, independence-assumption, or liberal?

8-4. (A) Suppose you want to handle or-combination of uncertainties nonnumerically. Suppose the possible degrees of uncertainty are "definitely", "probably", "probably not", and "definitely not".

(a) Suppose the "orcombine" function of any two of those four terms is defined by the table in Figure 8-5. Each row and column represent a pair of values to be combined. Which of the numerical or-combination methods is this equivalent to: conservative, independence-assumption, liberal, or something else?

(b) Suppose the "orcombine" function of any two of those four terms is defined by the table in Figure 8-6. Which of the numerical or-combination methods is this equivalent to: conservative, independence-assumption, liberal, or something else?

(c) For the method of part (b), suppose you want to map "definitely", "probably", "probably not", and "definitely not" into probabilities. It makes sense to have "definitely" = 1.0 and "definitely not" = 0.0. Give probabilities for "probably" and "probably not" consistent with the table in part (b). (There are an infinite number of answers.)

8-5. (A) Combination methods for probabilities use the Prolog **is**, which requires its arithmetic calculation to refer to only bound variables. Consider a rule-based system that uses rules with probabilities and calculates on those probabilities. Does the directionality of **is** mean that one of either backward chaining or forward chaining is impossible? Why?

8-6. (R,A) Consider the following way of doing or-combination of probabilities in an expert system: the cumulative probability is the fraction of the contributing probabilities that are greater than 0.5. So for instance the cumulative probability for contributing probabilities 0.3, 0.9, 0.66, 0.2, and 0.8 would be 0.6.

(a) Define a Prolog predicate **new_orcombine(PL,P)** that computes the cumulative probability **P** of a list **PL** using this method.

(b) Give two major disadvantages of this method, disadvantages not shared by the three methods discussed in the chapter, and explain why they are major.

8-7. Suppose we have R rules concluding D diagnoses such that there are the same number of rules concluding each diagnosis, |R / D|. Assume no intermediate predicates; have each diagnosis rules refer to facts. Suppose the probability of any rule succeeding in a random situation is P, and suppose this probability is independent of the success or failure of other rules.

(a) How many rules will succeed for a situation on the average?

(b) How many diagnoses will succeed for a situation on the average?

8-8. (G) Write 10 or so Prolog rules to predict what the weather will be at 3 P.M. some day, reasoning at noon that day, using probabilities as an additional argument to all predicates that have uncertainty. All rules should have a left side of the form **predict(<weather>,<probability>)**, where **<weather>** is either **sunny**, **partly_cloudy**, or **cloudy**. Choose a good evidence-combination method.

Assume the following predicates are available as the basis for your reasoning at noon. (Try to define some intermediate predicates based on these, which can then be combined to make predictions, to make things more interesting.)

> **current_west_view(<weather>)**: whether the given weather is viewable from a west-facing window right now. The argument can be **sunny**, **partly_cloudy**, or **cloudy**.
>
> **current_east_view(<weather>)**: same for east-facing window.
>
> **raining(<degree>)**: whether it is raining to that degree right now. The **<degree>** can be **light**, **steady_heavy**, and **cloudburst**.
>
> **weatherman_prediction(<weather>)**: whether the TV weatherman predicted that weather for 6 P.M. last night at 11 P.M.
>
> **grandma_prediction(<weather>)**: whether Grandma predicted that weather this morning from how her joints hurt.
>
> **grandma_memory**: whether you remember anything Grandma said that morning.
>
> **radio_prediction(<weather>)**: the weather predicted right now on the local radio news station.
>
> **secretary_has_radio**: you don't have a radio, but the secretary down the hall might.
>
> **secretary_out_to_lunch**: if they are out to lunch, the room is locked, and you can't get in.

8-9. (A,E) Consider the following argument: "Spending money on the lottery must be a good thing, because you constantly hear about people winning big bucks in the battery, in the newspapers and on television."

(a) What is the fallacy (reasoning error) here?

(b) What warning does this illustrate for using probabilities in rule-based systems?

8-10. Most students are adults (l8 years old or more). Most adults are employed. We could write:

```
adult(X,P) :- student(X,P2), P is P2 * 0.95.


employed(X,P) :- adult(X,P2), P is P2 * 0.9.
```

Then if we knew that Joe was a student with complete certainty, these rules say that Joe is employed with 0.855 probability, a clearly fallacious conclusion because most students don't work at all, as any professor will tell you. This fallacy is due to a simplification that we have made to make our analysis in the chapter easier.

(a) Show how the problem can be fixed by writing one rule as two.

(b) Suppose we consider this rule rewriting as awkward and we wish to fix things by just changing the probability combination method. The previous method was independence-assumption combination. What happens to the probability of Joe being employed when we use the conservative assumption?

(c) The conservative assumption does not give a reasonable answer either. How can we solve this problem in a reasonably general way?

8-11. (E) (a) In English, double negatives don't always mean what you expect them to mean. Explain why "not unhappy" is different from "happy".

(b) What warning does this suggest to the designer of a rule-based expert system using probabilities? In particular, for what sorts of English words should a designer be careful?

8-12. (P,G) Design a program to diagnose problems with cars. Many artificial intelligence applications involve diagnosis, and automatic diagnostic aids really help. We pick cars because almost everybody knows something about them, and we don't need to hire outside experts as with most expert systems. This diagnosis program should be usable by people who know virtually nothing about cars; it should use nontechnical words and familiar concepts. For instance, it shouldn't expect that a user knows what different sounds mean, and should try to describe sounds by analogies to everyday sounds.

An important part of the project will be the formal definition of concepts that may be concluded (e.g., "the car won't start"). Another part will be enumeration of things a user could be expected to know; in particular, try to use knowledge of the history of the car, what problems it has had in the past. Try not to get too technical; there are plenty of "common-sense" things about cars, particularly for the body and interior of the passenger compartment. For instance, one cause of a rattle in a car could be a baby's rattle under the seat.

Handle uncertain data and uncertain conclusions. Probably the easiest approach is independence-assumption combination. Decide initial values for probabilities, and how to treat evidence against something.

If this is a group project, one person should handle the control structure of the program and provide utilities for everyone else. Other people can specialize in different systems of the car. For instance, someone should probably handle the body and interior of the car, someone the electrical system, someone the engine and fuel system, etc. About thirty rules should be written by each contributor. Emphasize quality of the rules, not quantity.

Go to book index

Search

It's time for a change of pace. In this and the next two chapters, we'll cover a new class of artificial intelligence techniques. These are good for reasoning about events in time and plans of action. Technically, "search" | REFERENCE 1|. .FS | REFERENCE 1| Also called *heuristic search*, but that's misleading because you don't need heuristics to do it, and anyway the term "heuristic" is overused and much abused in artificial intelligence. .FE We'll need a new vocabulary to talk about search problems--terms like "state", "operator", "search strategy", "evaluation function", and "branching factor".

Search problems appear in many application areas. Search relates to rule-based systems because they involve actions occurring over time. In fact, search is an abstraction of the concept of a control structure. So we'll be looking at rule-based systems in a new way.

Changing worlds

So far the facts in Prolog databases have rarely changed. We added facts through caching in forward chaining, but those facts were logically derivable from the other facts and rules all along. We rarely removed facts from a Prolog database--only in implementing a few special programs.

But Prolog databases usually model a situation in the world, and the world can change. In fact, the result of running an artificial intelligence program may be to recommend actions in the world, which are then taken, and then reflected in changes to the facts true about the world. Often an artificial intelligence program is useless unless it does change something in the world. For instance, an automobile diagnosis program is not much help unless you can actually fix a car by its recommendations.

In many important applications, a problem can be solved only by a series of actions, not just one. Then the problem must track the changing situation in the world. Changing the facts appropriately can be complicated for such problems, and we need some new tricks.

States

A situation in the world described by some facts is a *state*. State is a fundamental concept in many areas of science and engineering. For instance, in physics a state is a set of measurable physical parameters fully describing some part of the universe; in simulation models of economics, a state is the set of values assigned to model parameters; and in computer systems programming, a processing state is a set of values currently in registers, buffers, and stacks. So a state is a "snapshot" of a process in time.

A state can be instantaneous or it can last for quite a while. In artificial intelligence (and computer science too) we only concern ourselves with non-instantaneous states, but where state changes (*branches* or *transitions*) are instantaneous. That is, state changes are *discrete* and not *continuous*. Loading a computer register is an example of a discrete state change. *Search* is the study of states and their transitions.

A state must include everything important about a situation in one bundle | REFERENCE 2|. .FS | REFERENCE 2| At least the states in this book. To make things easier, we deliberately ignore a class of searches performed using and-or-not lattices, for which states represent pieces of a problem. See other books under the topic of "searching and-or trees". .FE That means everything necessary to reason about it--and in artificial intelligence, that usually means a list of facts. These may be a lot of facts, and often we'll want to reason about many states, as when we try to figure how to achieve some desirable state. So two tricks often simplify state descriptions. First, we can keep only the facts relevant to the very specific problem we're working on. For instance, if our problem is to fix a car when we know there is something wrong with the electrical system, we can ignore facts about the state of the engine (whether it is cold or warm, whether the timing is accurate, whether the oil has been changed recently, etc.) since they don't matter. Second, we can compress the fact representation by collapsing several predicates into one, like the "relational database" predicates of Section 2.9. For instance, if ten screws hold the battery onto the chassis of a car, we can write one fact with ten arguments where argument number K is the word "on" or the word "off" reflecting the status of screw number K.

Three examples

Let's consider some examples of search problems. First, consider the problem of getting by car from one intersection to another in a city. There are different streets that you can take and different turns you can make. To complicate matters, some streets are one-way, and some turns are illegal. Also, you will probably want a short route rather than just any route. Considering this a search problem, a state s ithe position of a car in a city. We can represent this as the name of the intersection, since intersections are the only places where decisions must be made, and therefore the only locations that matter to reasoning. Solving a search problem means finding a series of states ending in some desired state, so the solution will be a sequence of intersections a car will traverse in order.

As a second example of search, consider the problem of car repair. Even if you know what's wrong, repair is rarely as simple as yanking the old part out and sticking the new part in. Usually you must unloosen fasteners (like screws and nuts) first, and move other parts out of the way to get to the part you want. You may need to run some tests on the suspect part to see if it is indeed faulty. Then you must reassemble the whole mess, doing the reverse of the previous steps in the reverse order--and sometimes exact reversal is impossible, as when removing a cover is easy but refastening requires careful hole alignment. You must also worry about the difficulty of various actions, if you want to get the job done efficiently; a sequence of actions that take half as much effort as another is better. For this problem, states are situations of partial disassembly (or assembly) of the car.

As a third example, consider the problem of smarter forward chaining for a rule-based expert system. Section 6.2 presented a simple general-purpose approach to forward chaining, using simple ideas like putting facts in a priority list in the order we find them (the focus-of-attention strategy). These ideas work for a wide class of applications. But for a specific application, a specific conflict-resolution policy may be better. For instance, we can keep the facts sorted by priorities of human experts (the experts' prestored hunches of relative importance), by our own analysis of the rules (like how many new conclusions a fact can lead to), by statistics on past runs of the rule-based system, or by dynamic features of the reasoning (like whether a certain fact has been proved yet). In general, intelligent forward chaining by intelligent fact sorting is a search problem with many options, each representing a fact to pursue, each leading to a different set of facts to choose from next, and so on. So a state for this problem is a set of unpursued (and perhaps also pursued) facts.

We'll use these examples subsequently in the chapter. They were picked to symbolize three important applications of search: optimization problems, human and robot task planning, and expert systems.

Operators

Our three examples involve many states. They can involve even more branches, ways directly from one state to another. It helps to group branches into categories called *operators*. Operators often correspond to verbs of English, the names of real-world actions.

Consider the three examples:

1. For finding routes in a city, the operators are possible actions at an intersection, since states correspond to intersections. So the operators are "go straight ahead", "turn right", "turn left", "turn into the first street to your right", "turn into the second street to your right", etc.

2. For car repair, the operators are much more numerous, including every possible action that can be done on the car: unfasten, unscrew, remove objects, manipulate objects, replace, align, fasten, screw, and so on.

3. For smarter forward chaining for a rule-based system, there seems to be only one operator: remove a fact from the set of unexamined facts and pursue its implications.

Be careful though, because operators are not well-defined for many problems. We could, for instance, consider each screwing-in for every single screw as a separate operator; or we could consider each pursuit of a fact as a separate operator. But if operators get too specific, they lose their chief advantage of abstraction over many state transitions.

Preconditions and *postconditions* are frequently associated with operators. Preconditions for an operator are things that must be true before the operator can be applied, writable as predicate expressions. For instance, a precondition to the operator of removing the battery from a car is that all the screws fastening the battery to the chassis are removed. Analogously, postconditions are conditions that must be true after an operator has been applied. For instance, a postcondition of removing the battery from a car is that the car can't start when the ignition switch is turned. Preconditions are useful because they suggest what is necessary to apply an operator; postconditions are useful because they summarize the effect of an operator.

Search as graph traversal

The main reason for identifying states and operators is to let us plan in advance a solution to a search problem. By reasoning hypothetically about operator sequences, we can avoid blundering about in the real world. Planning can also ensure that we find the *best* solution to a problem, not just any solution.

For such advance planning, we can consider a search problem as a task of traversing a directed-graph data structure. Semantic networks (see Section 2.7) are one kind of directed graph important in artificial intelligence, but there are several others too. Graphs consist of nodes (small named circles) and directed edges (named arrows) connecting nodes; we can equate nodes with states, the names on directed edges with operator names, and edges themselves with branches or operator applications. In a search problem, we are given a *starting* state and one or more finishing or *goal* states. On the search graph this means finding a path or traversal between a start node and one of a set of goal nodes. (Here we broaden the term "goal" beyond its meaning for rule-based systems, in which a "goal" is a predicate expression we want to prove; in search, a "goal" is any desired final state.)

Figure 9-1 shows an example route-planning problem. As we said, city-route planning is a search problem for which the states are intersections (intersections are the only places where we must make choices). Then a search graph can be drawn as in Figure 9-2. There is just one goal state.

Figures 9-3 and 9-4 show similar example search graphs for the auto repair and the smarter forward chaining problems. The latter is interesting because we don't know the goal state in advance: we just keep searching until there are no more facts to pursue, and that's a goal state. If we knew what a goal state was in advance, we wouldn't need to do any work. An important class of search problems similarly defines goal states indirectly, in that they only can recognize one when they see one.

The simplest search strategies: depth-first and breadth-first

There are many control structures for search; these *search strategies* are summarized in Figure 9-5. The two simplest are depth-first search and breadth-first search.

With depth-first search, the start state is chosen (*visited*) to begin, then some *successor* (a state that we can reach by a single branch or state transition) of the start state, then some successor of *that* state, then some successor of that, and so on until we reach a goal state. Usually the choice among successors of a state isn't arbitrary, but made by "heuristics", which we'll explain in Section 9.7. If depth-first search reaches a state S without successors, or if all the successors of a state S have been chosen (visited) and a goal state has not yet been found, then it "backs up". That means it goes to the immediately previous state or *predecessor*--formally, the state P whose successor was S originally. Depth-first search then takes the next-suggested successor choice of P. So "backing up" is just like Prolog backtracking.

An example will make this clearer (see Figure 9-6). The circled letters are states, and as before, the arrows are branches (operator applications). Suppose S is the start state and G is the only goal state. Suppose that vertical height of the state on the page is used to rank states when a choice is necessary, with higher states having higher rank. Depth-first search will first visit S, then A, then D. But D has no successors, so we must back up to A and try its second successor, E. But this doesn't have any successors either, so we back up to A again. But now we've tried all the successors of A and haven't found the goal state G, so we must back up to S.

Now S has a second successor, B. But B has no successors, so we back up to S again and choose its third successor, C. C has one successor, F. The first successor of F is H, and the first of H is J. J doesn't have any successors, so we back up to H and try its second successor. And that's G, the only goal state. So we're done. The solution path to the goal is S, C, F, H, and G.

One problem with depth-first search is that it works fine when search graphs are trees or lattices, but can get stuck in an infinite loop on other graphs (see Appendix C for definitions of these terms). This is because depth-first search can travel around a cycle in the graph forever. One fix is to keep a list of states previously visited, and never permit search to return to any of them. We will do this in our depth-first program in the next chapter, but this check can require a lot of time since there may be lots of previous states, and states with long, complicated descriptions.

Breadth-first search does not have this danger of infinite loops. The idea is to consider states in order of increasing number of branches (*level*) from the start state. So we first check all the immediate successors of the start state, then all the immediate successors of these, then all the immediate successors of those, and so on until we find a goal state. For each level, we order states in some way as with depth-first search. For Figure 9-6, S is on level 0; A, B, and C are on level 1; D, E, and F, level 2; H and I, level 3; and J, G, and K, level 4. So breadth-first search, assuming the previously-used vertical ordering among same-level states, will consider in order S, A, B, C, D, E, F, H, I, J, and G--and then stop because it's reached the goal state. But the solution path it found was the same as depth-first's.

Breadth-first search is guaranteed to find a goal state if a path to one exists, unlike depth-first, but it may take a while. Loops in the search graph will cause inefficiency (useless extra paths being considered) instead of infinite processing loops. If that inefficiency is bothersome, we can use the same trick in depth-first of storing a list of previously visited states and checking against it. Notice that by working by level, any later path we find to a state can be no shorter than the first path found to it, so the first path found to a state can be considered the best.

Depth-first and breadth-first search often occur in disguise in artificial intelligence and computer science. The backward chaining, forward chaining, and rule-cycle hybrid chaining algorithms of Chapter 6 were really kinds of depth-first search, so they can be called *depth-first control structures*. Depth-first control structures are common in computer applications because of their intimate connection to stacks, an easy-to-implement data structure. Though we didn't discuss it in Chapter 6, we could do breadth-first backward chaining as a variant control structure, or breadth-first forward chaining. So be careful not to confuse the backward/forward chaining distinction with the depth-first/breadth-first search distinction--they're quite different things.

Heuristics

Depth-first and breadth-first search are easy to implement but often inefficient for hard problems. A top concern of artificial intelligence research has been finding better search strategies. Many of these better strategies are related to depth-first and breadth-first search. Two common variants are search using heuristics and search using evaluation functions.

Heuristics are any nonnumeric advice about what order to try the successors of a state for further search | REFERENCE 3|. .FS | REFERENCE 3| Some authors say heuristics can be based on numeric calculations, but we'll call such things *evaluation functions* to prevent confusion. .FE So their effects are "local": they give advice for specific successor choices, not about the whole search strategy. Since both depth-first and breadth-first must make such choices, both can use heuristics, as well as the other search strategies we'll discuss in this chapter. Heuristics are like gardeners because they *prune* (eliminate) branches. Heuristics are a generalization of meta-rules (see Section 6.7), rules about rules.

Usually heuristics are not guaranteed--that is, they represent reasonable advice about how to proceed, and may be wrong. But if they're wrong a lot, there's no point in using them. Heuristics need not give a unique recommendation of what's best in every situation. At worst, we can choose at random among multiple recommendations. What if that's not a good idea? "Search" me.

Here are example heuristics:

> --For city route planning, never turn right twice in a row, since this tends to make you go back in the direction you came from.
>
> --For city route planning, turn whenever you find you've left city limits.
>
> --For car repair, never remove a part unless it is near another part you think is faulty.
>
> --For car repair, take out small parts first.
>
> --For smarter forward chaining, pursue facts that occur together with other known facts on the right side of a rule.
>
> --For smarter forward chaining, pursue facts that led to useful conclusions on the ten most recent runs.

Heuristics are not all equally valuable. Compare:

1. For car repair, do little jobs first.

2. For car repair, first take out small parts attached to objects you want to fix.

3. For car repair, first take out screws attached to objects you want to fix.

4. For car repair, if you must fix the alternator, take out its mounting screws first.

5. For car repair, if you must fix the alternator, take out its Right Front Mounting Screw first.

These heuristics span a spectrum from general to specific. Really general advice like #1 is hard to apply (it's hard to decide what's a "little job" and what isn't) and wrong in many cases (it would recommend equally the removal of any screw in the car, most of which are irrelevant). Such heuristics are "proverbs", like "Haste makes waste"; they sound nice, but they're nearly useless. At the other extreme, heuristic #4 and #5 is too specific to be helpful: it just applies to one action involving one screw in a car, and its effect could be had by just putting conditions on the "fix-alternator" operator. And if we tried to use heuristics like #5 in a search problem, we'd need a lot of them--probably many more than the number of operators--so search efficiency would probably decrease, not increase as is the purpose of heuristics. Our best bets are heuristics like #2 and #3 that compromise between extremes; probably #3 is better because "small part" is hard to define.

Evaluation functions

A big difficulty with heuristics is disagreements between them, situations for which two heuristics make contradictory recommendations. If one heuristic is known to be better than the other, then it should have priority, but the two can seem equally good. If such difficulties arise frequently, it's better to rate states numerically. A method for calculating such numbers is called an evaluation function. By convention the values of evaluation functions are nonnegative numbers such that the smaller the number, the better the associated state; and goal states have an evaluation function value of zero. Evaluation functions can be in any form and can use any available information about a state; they can also use a description of the goal states. (However, it's desirable that evaluation functions be "smooth"; that is, if they're calculated using numbers, they shouldn't ever jump abruptly in value when the numbers vary slightly.)

Some example evaluation functions:

> --For city route planning, take the straight-line ("as the crow flies") distance between an intersection and the goal intersection (that is, prefer the successor state closest to the goal state along a straight line).
>
> --For city route planning, take the straight-line distance to the goal plus one tenth of the number of streets crossed by that straight line (this helps you avoid stop signs and traffic lights).
>
> --For car repair, take the number of parts removed from the car plus the number of faulty parts in the car (this will increase in the first half of the solution, but will decrease in the last half, and should be kept small anyway).
>
> --For smarter forward chaining, take the number of right-side expressions in rules minus the number of facts (proved and given) in some state. (This isn't likely to approach zero, but it does guide search helpfully.)

Evaluation functions make possible two new search strategies, summarized on the third and fourth rows in Figure 9-5. The evaluation-function variant of depth-first search is called *hill-climbing* (or sometimes *discrete optimization*) search; the evaluation-function variant of breadth-first search is called *best-first* search. However, best-first usually has an additional twist beyond breadth-first: the best-evaluation (lowest-evaluation) state of those *anywhere* in the search graph, of those whose successors have not yet been found, is picked, not just a state at the same level as the last state. So best-first search usually "jumps around" a lot in the search graph, always picking the minimum-evaluation state of all the unvisited states it knows about.

To illustrate best-first search, take the search graph of Figure 9-6 and assume the evaluation function values shown as circled numbers in Figure 9-7: S:12, A:7, B:8, C:8, D:6, E:4, F:7, H:4, I:5, J:2, G:0, and K:1. (Ignore the numbers in *squares* and the numbers beside the arrows for now.) As before, assume S is the starting state and G is the only goal state. Best-first search would start with S, and would find and evaluate its successors A, B, and C, to discover that A is the minimum-evaluation one. So A is picked next, and its successors D and E evaluated. The states not yet examined to find successors (*visited*) are D, E, B, and C, with evaluation function values 6, 4, 8, and 8 respectively. E is the best, but it has no successors; D is the second best, but it has no successors.

We've got a tie between the two remaining unexamined states, B and C. In such cases, heuristics can decide (and perhaps for near-ties too). Assuming the vertical ordering heuristic used with the depth-first and breadth-first examples in Section 9.6, B should be picked next. But it has no successors, so C must be picked. It has one successor, F. F has two successors, H and I, with evaluation function values 4 and 5. The 4 is better, so H is better, and is picked next. H has successors J, G, and K with evaluation function values 2, 0, and 1. But G is a goal state, so we can stop. In summary, best-first search examined the states in the order S, A, E, D, B, C, F, H, G. That's different from both depth-first and breadth-first.

Cost functions

Evaluation function values are not the only numbers helpful in search. For some problems, we don't want just *any* path to a goal state, but a good or best path. We could assign costs to each operator, total up these costs along solution paths, and rate the paths. This requires a nonnegative *cost function* measuring something like the difficulty by the sums of going from one state to another.

Some example cost functions:

--For city route planning, the total length in meters of a path;

--For city route planning, the number of intersections in a path (since intersections slow you down);

--For car repair, the time in minutes needed to do a sequence of actions;

--For car repair, the amount of energy in calories needed to do a sequence of actions;

--For smarter forward chaining, the computer time in seconds required to pursue facts in a particular order;

--For smarter forward chaining, the amount of main-memory storage in bytes required to pursue facts in a particular order.

Cost functions and evaluation functions are easy to confuse. Remember that evaluation functions refer to the future, cost functions to the past. That is, evaluation functions guess how close a state is to a goal state, while cost functions measure how far a state is from the start state. So cost functions are more concrete than evaluation functions. A cost function often suggests an associated evaluation function, not the other way around, because (as we will see) it is useful to have them in the same units. So if a cost function measures in meters, its evaluation function should too.

Optimal-path search

A search that must find the lowest-cost (*optimal*) path to a goal state, instead of just any path, needs a different search strategy from those so far considered. If we have a cost function but no good evaluation function and no good heuristics, we can use *branch-and-bound* search (see Figure 9-5). It's like best-first search but using costs instead: it always finds successors of the state whose path has lowest total cost from the start state. Such a strategy may "jump around" among states as best-first search does, but it has a nice property: the first path to the goal that it finds is guaranteed to be the lowest-cost path to the goal.

If we have both cost and evaluation functions, we can use an *A* search* strategy (that's pronounced "A-star"). The idea is to sum the cost and evaluation function value for a state to get a measure of overall worth, and use these numbers to select states in a best-first search algorithm instead of just the evaluation function values. (This sum makes most sense when the cost function and evaluation function are in the same units.) So A* search is sort of a hybrid of best-first (using an evaluation function) and branch-and-bound search (using a cost function), incorporating information from both, and often giving better performance than both. As with branch-and-bound search, a certain guarantee applies to a solution found by A* search: if the evaluation function value for any state S is always no more than the subsequently found cost from S to the goal, then the first path to the goal found by A* search is the lowest-cost path to a goal. But A* is often still a good search strategy even when the guarantee doesn't hold.

Suppose we use the A* strategy instead of best-first on the search graph of Figure 9-7. Recall that the numbers in circles are the evaluation function values of states. Suppose the costs for path segments are shown by the numbers next to the arrows: 4 for A to S, 1 for A to D, 2 for A to 3, 1 for S to B, 2 for S to C, 1 for C to F, 3 for F to H, 1 for F to I, 1 for H to J, 2 for H to

G, and 2 for H to K. The criterion number for a state is now the sum of the evaluation function value and the costs along the path leading to that state. For instance, the criterion number for state H is |4 + 3 + 1 + 2 = 10|. The other criterion values are: S:12, A:11, D:11, E:10, B:9, C:10, F:10, I:9, J:9, G:8, and K:9; these numbers appear inside squares in Figure 9-7. A* search will work like best-first but use the numbers in the squares instead of the numbers in the circles, and it will visit the states in the order S, B, C, F, I, H, and G. (The evaluation function values are not a lower bound on the cost to the goal, but that doesn't matter when there's only one path to the goal.)

A route-finding example

To better illustrate the differences between search programs, we show results from a program for one of our three standard examples, finding city routes. Figure 9-8 shows a portion Monterey, California, USA for which we stored street-intersection coordinates. The problem was to go from the the point marked "start" to point marked "goal". The evaluation function was the straight-line distance to the goal (computed from approximate coordinates of each intersection), and the cost function was the distance along the route (computed by summing straight-line distances between successive intersections). Shown on the map are the results of three search strategies: breadth-first, best-first, and A* search. As you can see, the paths are different: A* finds the shortest path, breadth-first minimizes the number of intersections, and best-first tries to keep moving towards the goal. (Though not shown on the map, depth-first search wanders around stupidly.)

Special cases of search

Certain tricks sometimes make search problems easier. One trick often possible is reversing the search, working backward from goal states to start states. For instance:

> --for city route planning, find a path from the destination to the start;
>
> --for car repair, solve the first half of the job by reasoning from the part P that must be fixed, deciding what other part P2 needs to be removed to get to P, what other part P3 must be removed to get to P2, and so on.

This does require you to know beforehand all goal states, something not possible for the smarter forward chaining problem and other important problems, in which finding a goal state is the whole point of the search (though backward chaining does represent something like a reverse of forward chaining). If you know more than one goal state, you can either try reverse search with each in turn, or make all the goal states the starting set of "unexamined" states for those strategies that use them.

Backward search generally requires different *reverse* operators from the forward search operators. For instance in city route planning, the reverse of a right-turn operator is a "backward left-turn" operator. So the solution to the backward search can be different from the solution to the forward search even if the same heuristics, evaluation function, and/or cost function are used. Fortunately in many search problems, the backward operators *are* identical to the forward operators, as in route planning on a larger scale, planning of routes between cities.

Whenever backward search is good, an even better idea is parallel forward and backward search on the same problem--*bidirectional search*. Run both searches independently (either on two processors or time-sharing on a single processor), and stop whenever they meet (or reach the same state). As illustration, compare the top and middle diagrams in Figure 9-9. Bidirectional search often finds a solution much faster than either forward or backward search alone, because the number of search states usually increases quickly as we move farther from the starting point, and with two searches we're searching half as deep. But it does have a serious danger: if we use a poor search strategy for both searches, like best-first with a poor evaluation function or, often, or depth-first search, the two searches may "bypass" one another instead of meeting in the middle.

Another trick we can often use to make search easier is *decomposition*, breaking the search problem into several simpler sub-searches, each of which can be solved independently. This usually means finding an intermediate state through which the search must go | REFERENCE 4|. .FS | REFERENCE 4| A starting state usually has several branches from it. But each branch isn't considered a "decomposition" of the problem--the whole search problem isn't made any simpler by looking at it this way, as it usually is by decomposition around intermediate states. .FE Then we solve two separate search problems: getting from the start to the intermediate state, and getting from the intermediate state to a goal state. As illustration, compare the top and bottom diagrams in Figure 9-9. You can also decompose a problem into more than two pieces, if you can figure out more than one intermediate state that the search must go through.

For city route planning, if the goal is a place on the other side of a river, and only one bridge crosses the river, you must use that bridge. So you can simplify the problem by decomposition into two subproblems: getting from your start to the bridge, and getting from the bridge to the goal. For car repair, a good decomposition is into three pieces: getting the faulty part out, fixing the faulty part, and then reassembling the car. Decomposability of search depends on the problem, but whenever it's possible it's usually a good idea because shallow search problems can be a lot easier than deep search problems.

Another useful feature of a search problem is *monotonicity*. If whenever an operator can be applied from a state S, that same operator can be applied from any state reachable from S (by a sequence of successors), then the search problem is monotonic. Often we don't need to be so careful in choosing an operator in a monotonic search problem, because we can can get much the same effect by applying the overlooked operator later, and hence we don't need terribly good evaluation functions and heuristics. But monotonic search problems aren't common. Only the third of our three standard examples is monotonic, the smarter forward chaining problem (since a postponed pursuit of some fact F is always doable later). Many searches for proving things are monotonic, since provable things usually don't stop being provable later (what is called *monotonic logic*). One word of warning: monotonicity depends on the operator definitions, and different operators can be defined for the same search problem, so a search problem monotonic for one set of operators may not be for another.

How hard is a search problem?

It helps to know the difficulty of a search problem before tackling it so you can allocate time and space resources. Sometimes what seems an easy problem can be enormously difficult, or a problem very similar to a really difficult one can be easy. So people try to estimate the number of states that need to be studied to solve a search problem. Two methods can do this: bounding the size of the search space, and calculating the average *branching factor* or *fanout* of successive states.

The first method takes an upper bound on the number of states that need to be examined to solve a problem as the number of possible states in the entire problem. This set of all possible states is called the *search space*, and what's needed is the size of the search space. (Don't confuse the term "search space" with the amount of memory needed to search: "space" is used abstractly here, the way mathematicians use "vector space".) For some problems, the size of the search space is easy to see from the description of the problem, as for city route planning in which it's the number of intersections in the city, something we can count. For other problems, we can use what mathematicians call *combinatorial methods*. For instance, for car repair we can describe the condition of every part in the car as either in the car and faulty, in the car and OK, out of the car and faulty, or out of the car and OK. So if there are 1000 parts in the car, the size of the search space is 4 to the 1000th power--a lot! That means heuristics or an evaluation function are necessary to solve this problem, because we'll never succeed if we try operators at random.

The size-of-the-search-space method of estimating search difficulty can't always estimate the size of the search space so easily, as in smarter forward chaining when we can't tell in advance which or even how many facts we'll prove. Also, the method calculates only an upper bound on difficulty. Alternatively, we can reason how the number of states increases with each level of search, and figure how many levels deep we'll go. The number of successors (usually, previously unvisited successors) of a state is called its *branching factor*. (Caution: don't confuse the branching factor with the number of *operators* that can generate successors, a smaller number.) If the branching factor doesn't differ much between states, we can speak of an *average branching factor* for the whole search problem. Then we can estimate the number of states at level K in the search graph as |B sup K|, B the average branching factor. Notice that this exponential function gets large very fast as K increases--the so-called *combinatorial explosion*. So if we can estimate the level of a goal (the number of states from the starting state to it), we can estimate the number of states that will be visited by breadth-first search, summing up the estimate for the number of states at each level. This is also a good approximation for best-first search, branch-and-bound search, and A* search, when evaluation and cost functions aren't very helpful in guiding search.

As an example, the average branching factor for the city route problem is around three, since there are usually three branches (directions of travel) at an intersection: go straight, turn right, and turn left. Suppose we know we are about 10 blocks from where we want to go. Then for a breadth-first search there are approximately three states at level 1, nine states at level 2, 27 at level 3, and so on up to |3 sup 10| for level 10. So the total number of states of level 10 or less is approximately the sum of a geometric series:

```
1 + 3 + 9 + 27 + ...   =   ( 3 sup 11 - 1 ) / ( 3 - 1 ) = ( 3 sup 11 - 1 ) / 2   =
88573
```

In general, in a search problem with an average branching factor of B, the number of states up to and including those at level K is |(B sup {K+1} - 1) / (B - 1)|. If B is large (say 5 or more), this can be approximated by |B sup K|, the number of states at the deepest level. These formulas help quantify the advantages of bidirectional search and search decomposition. If you divide a problem whose solution is |R| states long into two approximately even halves, each half is about |R / 2| states long. If the branching factor B is large, and the same in both directions, the number of states examined in the two half-problems is much less than the number of states examined without decomposition or bidirectional search because:

```
B sup { R / 2 } + B sup {R / 2 }   "is much less than"   B sup R
```

because, dividing both sides by |B sup {R / 2}|:

```
2   "is much less than"   B sup {R / 2}
```

The effect of heuristics and evaluation functions is to rule out certain successors for particular states in preference to other successors. So in effect, heuristics and evaluation functions decrease the average branching factor. This means we can get farther down into the search graph faster. So the usefulness of heuristics and evaluation functions can be quantified as a ratio of average branching factors with and without them.

Backward chaining versus forward chaining (*)

Analysis of search-problem difficulty lets us quantify for the first time differences between the control structures discussed in Chapter 6. Take for instance this rule-based system:

```
t :- a, b.
t :- c.
u :- not(a), c.
u :- a, d.
v :- b, e.
```

And further assume the facts **a**, **e**, and **d** only are true, and given in that order.

Backward chaining will try the first three rules, which fail, and then the fourth, which succeeds. Basic backward chaining without caching involves eight total queries: top-level predicates **t** and **u**, **a** (which succeeds), **b** (which fails), **c** (which fails), **a** again (which succeeds), **a** again, and finally **d** (which succeeds). The time to do this is approximately proportional to the number of queries made because each query requires an index lookup of nearly-equal time.

Pure forward chaining will match fact **a** to expressions in the first and fourth rules, then match **e** to an expression in the last. Finally, it matches **d** in the fourth rule, and **u** is proved. So four matches are made. The time to do this is approximately proportional to the number of matches because each requires an index lookup of nearly-equal time. Now we just compare eight queries to four matches to decide whether backward or forward chaining is better for this very specific situation. Though this is a comparison of different units ("apples with oranges") we can usually conduct experiments to equate both queries and matches to time units.

The main problem with this analysis is that it's hard to generalize: we may not know which or how many facts are going to be true in a situation, and there are many possible situations. Simulations are one approach. But we can also use probabilities in mathematical analysis, as we'll now show.

Let's consider backward chaining first, still using the previous example. Suppose that potential facts **a**, **b**, **c**, **d**, and **e** have independent probabilities P of truth. Then the probability of the first rule **t :- a, b.** succeeding is |P sup 2|, the probability of the second succeeding is |P|, the third |(1 - P) P|, and the fourth and fifth |P sup 2|. The expected number of queries generated when the rules are in this order is a long, tricky formula because we don't need to fully evaluate a rule to have it fail:

```
3 P sup 2 + 3 ( 1 - P ) P + 4 P ( 1 - P ) P + 8 P ( 1 - P ) sup 2 P

    + 10 ( 1 - P ) P ( 1 - P ) P + 10 P ( 1 - P ) sup 3
+ 9 ( 1 - P ) sup 3 + 10 ( 1 - P ) P ( 1 - P ) sup 2
```

We got this formula by reasoning about the search lattice, which was actually more like a decision lattice, for the situations possible (see Figure 9-10). Here are the rules again:

```
t :- a, b.
t :- c.
u :- not(a), c.
u :- a, d.
v :- b, e.
```

The first term in the formula corresponds to success of the first rule, terms 2 and 3 to the success of the second, no terms to the third (there's no way the third rule can succeed given the second fails), term 4 to the success of the fourth rule, term 5 to the success of the fifth, and the remaining terms to the three different cases in which all the rules fail. To get this formula, we separately considered cases with **a** true and with **a** false.

We can reason similarly about forward chaining when facts are always taken in an order and each has the same probability P of occurrence (see Figure 9-11). Suppose the order is **a** before **b**, **b** before **c**, **c** before **d**, and **d** before **e**. The formula for the number of fact-rule matches is:

```
1 P ( 1 - P ) + 3 P sup 2 + 3 P sup 2 ( 1 - P ) + 3 P sup 2 ( 1 - P ) sup 2
+ 4 P sup 2 ( 1 - P ) sup 3 +  3 P ( 1 - P ) sup 4
+ 4 P sup 3 ( 1 - P ) sup 2

+ 4 P sup 2 ( 1 - P ) sup 3 + 3 P sup 2 ( 1 - P ) sup 3
+ 3 P ( 1 - P ) sup 4 + 3 P sup 2 ( 1 - P ) sup 3 + 2 P ( 1 - P ) sup 4
+ 2 P ( 1 - P ) sup 4 + 1 ( 1 - P ) sup 5
```

Knowing the ratio of the cost of a backward-chaining query the cost of a forward-chaining match, we can compare the two formulas for a specific value of P to decide whether backward or forward chaining is better.

Using probabilities in search (*)

Since probabilities are numbers, they can guide hill-climbing, best-first, branch-and-bound, and A* searches. For instance, the probability that a state is on a path to a goal can be used as an evaluation function. Or in search during forward chaining, facts can be ranked for selection by one million minus the reciprocal of their a priori probability, so unusual facts are followed up first. In backward chaining using rules with uncertainty like those in Chapter 8, rules can be selected by the reciprocal of their rule strength (the rule strengths being the probability of their conclusion under the most favorable conditions.)

Another example: visual edge-finding as search (*)

Computer vision is an important subarea of artificial intelligence. It's complicated, and its methods are often quite specialized, since two-dimensional and three-dimensional information seems at first to be quite different from predicate. But it also exploits many of the general-purpose techniques described in this book, including search.

One role search plays is in edge finding and edge following, as a kind of "constructive" search that actually builds something as it performs a search. Most computer vision systems start from a digitized image, a two-dimensional array representing the brightness of dots in a television picture. The picture is first "cleaned up" and "smoothed out" to make it easier to analyze, using some mathematical tricks. Any remaining sharp contrasts between adjacent parts in the picture are important--contrasts in brightness, color, and the "grain" or *texture* of small adjacent regions. These contrasts can be used to make a line drawing (a drawing consisting only of lines) of the picture, where lines correspond to boundaries of high contrast between regions of mostly-homogeneous characteristics; the lines are called *edges*. So a line drawing can be a kind of data compression of the original picture. Line drawings provide a basis for most visual analysis techniques, techniques that try to figure out what the picture is showing.

But edge-finding isn't as easy as it may seem. The problem is that for various reasons, things in the real world that you'd think would make edges don't. Consider Figure 9-12. Different surfaces may coincidentally be the same brightness and color along part of their edge, or an edge may lie in shadow, or glares and reflections may cover the edge, or the resolution of the picture may be insufficient to pick up small details of edges. So usually only some edges and parts of edges in a picture can be recognized, meaning line drawings with gaps. Human beings can easily fill in these gaps, because they have strong expectations about what they will see and their vision automatically fills in details. But computers must be taught how.

A good first step is to quantify the "edgeness" of each dot in the picture. Several mathematical formulas can be used, but we'll show here a simple one that only examines the brightness of its cell and its immediate neighbors. (Edges between regions of different color can be found by looking at brightness of the picture viewed through colored filters.) Suppose we represent the picture as a two-dimensional array |b (i , j)| of numbers representing light intensities. Then the *magnitude of the gradient* for each dot is defined as:

```
g ( i , j )   =   sqrt { ( b ( i+1 , j ) - b ( i-1 , j ) ) sup 2 +
( b ( i , j+1 ) - b ( i , j-1 ) ) sup 2 }
```

This is a measure of "edgeness" for every dot. The larger this number is, the more the brightnesses around some dot in the picture differ among themselves.

Now we're ready to formulate edge finding as a search problem. A state can be represented as a two-dimensional bit array |e (i , j) | (that is, an array of things with Boolean or true/false values) with an entry for every dot in the picture. A "true" means that in the corresponding picture, the dot lies on an edge; in the starting state, every dot is marked "false". A branch between states changes a single element from "false" to "true", meaning we've decided an edge is there. There's only one operator: mark |e (i , j)| as "true" for some i and some j. Figure 9-13 gives an example interpretation problem. The upper part shows an array of edgeness measures |g (i , j)|, and the lower part shows a reasonable assignment of edges |e (i , j)| to the corresponding cells.

However, there is no well-defined goal for this problem. It's more like an optimization problem in operations research. We want to maximize the number of "good" edges identified, so it sounds a bit like a best-first search with an evaluation function. Several things can be considered in an evaluation function:

> --the number of cells with edgeness more than some number C, of those for which |e (i , j)| is false (this measures the completeness of the edge assignments);
>
> --the total number of approximately straight line segments formed by the dots marked "true" (this measures the "elegance" of the edge assignments);
>
> --the sum of the average curvatures for all such approximately-straight line segments (this measures straightness of segments);
>
> --the number of true-marked cells with exactly two true-marked immediate neighbors (this measures narrowness of edges);
>
> --the negative of the average edgeness measure of all true-marked cells (this measures the selectivity of true-marking).

Whenever several different measures like these describe progress in a search problem, a good idea is to take a weighted average of these, and make that the evaluation function. (Or if all the numbers are positive, take the product.) That seems good here, but we would need to experiment to find the proper weightings.

Despite the lack of a goal per se, it helps to invent one to prevent working forever. A simple but effective criterion is to stop at state S if no successor state under consideration is better than D worse than the evaluation of S, for some fixed constant D.

Notice how this search problem differs from city-route planning: the edge finder will rarely mark adjacent cells in succession. In fact, it's likely to jump around considerably, since a good heuristic is to mark the cell with the highest "edgeness" among those not yet marked "true". With that heuristic, we're unlikely to mark adjacent cells in succession. This nonlocality of processing appears in many different vision applications, and reflects the concurrency apparently present in much human vision.

.SH Keywords:

search
state
branch
operator
precondition
postcondition
starting state
goal state
search strategy
depth-first search
breadth-first search
hill-climbing search
best-first search
heuristic

pruning
evaluation function
cost function
optimal-path search
branch-and-bound search
A search*
backward search
bidirectional search
decomposability
monotonicity
monotonic logic
search space
branching factor
combinatorial explosion

Exercises

9-1. (E) Explain why none of the following are states as we define the term for search.

(a) All the important events that happen in an organization in one day.

(b) The current temperature of a room.

(c) The mental condition of a person's mind.

9-2. (A,E) Explain how using inheritance to find a value of the property of some object is a kind of search.

9-3. The search graphs in this chapter have all been *planar*--that is, none of the branches crossed. Is this necessarily true of all search graphs?

9-4. (a) Modify the description of the algorithm for pure forward chaining, given in Section 6.2, to work in a breadth-first way.

(b) Explain how a breadth-first backward chaining might work.

9-5. Consider the problem of driving by car from one place in a city to another. Assume you do not know the way. Consider this as a search problem with three operators:

A: travel to the next intersection and keep going straight, if legal
B: travel to the next intersection and turn right, if legal
C: travel to the next intersection and turn left, if legal

(a) Suppose you are driving a car without a map. Which control strategy is better, depth-first or breadth-first? Assume you can use simple heuristics (like "turn around if you're in the country") to stay near the goal.

(b) Suppose you planned a route with a map, and the map was slightly wrong (for instance, a street is closed for repairs). How could you prepare to better handle such occurrences?

9-6. Consider the complete search graph for some problem shown in Figure 9-14. Suppose state "a" is the starting state, and the states shown are the only possible states (and none of the states shown is the goal). Numbers written next to states represent the evaluation function at those states, and numbers written next to branches represent the cost function on those branches. (Some of this information may be irrelevant.)

(a) Which is the fourth state whose successors are attempted to be found by a depth-first search using the heuristic that states whose names are vowels (a, e, i, o, or u) are preferred to states whose names are not?

(b) Which is the fourth state whose successors are attempted to be found by a best-first search?

9-7. (a) For the search graphs of Figures 9-6 and 9-7 we imposed a heuristic that states be ordered when necessary by vertical height on the page from top to bottom. Explain how this can be considered either a heuristic or an evaluation function.

(b) An evaluation function is just a way to assign nonnegative numbers to states (not necessarily a continuous function on any variable). Explain how an evaluation function can always be constructed to duplicate the effect of any set of heuristics for a search problem.

(c) Now consider the reverse direction. In a search problem, can you always go from an evaluation function for that problem to a finite set of (nonnumeric) heuristics that have the same meaning?

9-8. (E) (For people who know something about music) Explain how harmonizing a melody is a kind of search. What are the states and operators? What are some good heuristics? Why is backing up to previous states necessary?

9-9. (a) Consider A* search in which the evaluation function is zero for every state. What is the name for this search?

(b) Consider A* search in which the cost function is zero for every state. What is the name for this search?

(c) Consider A* search in which the evaluation function is zero for every state, and the cost function is the length of the path to the state. What is the name for this search, besides the name for part (a)?

9-10. (A) Suppose for some search problem for which you want to use the A* search you have found an evaluation function that never overestimates the cost to a goal state by more than K units. How can you get a guaranteed-optimal solution from A* search?

9-11. (R,A) Suppose for a search problem there are three operators, Op1, Op2, and Op3. Suppose in the starting state you can apply any of the three. Then suppose if the first operator was not Op3 you can apply a different operator for the second action than the first operator. Assume no other operator applications are possible. No goal is given, so a search must eventually explore every possible state.

(a) Suppose you do breadth-first search using the heuristic that Op1 branches are preferred to Op2 branches, and Op2 to Op3. Draw the state diagram, and label the states in the order in which you try to find their successors. Use labels a, b, c, d, e, f, g, and h.

(b) Suppose you do best-first search for which the evaluation function is

6 after Op1 then Op2
4 after Op1 then Op3
9 after Op2 then Op1
11 after Op2 then Op3
8 after Op1
7 after Op2

5 after Op3
10 for the starting state

List the states in the order in which you try to find their successors.

(c) Suppose you do A* search for which the evaluation function in part (b) and the cost function is

2 for Op1
5 for Op2
9 for Op3

List the states in the order in which you try to find their successors.

9-12. (H) Suppose we have computer terminals we wish to move to different floors of a three-floor building. Suppose at the start:

1. one terminal on the third floor belongs on the second floor;
2. one terminal on the second floor belongs on the first floor;
3. two terminals on the first floor belong on the third floor;
4. another terminal on the first floor belongs on the second floor.

In the starting state, the elevator is on the first floor. In the goal state, each terminal is on the floor where it belongs. Assume there are two operators:

A: take one terminal from floor X to floor Y, X different from Y;
B: take two terminals from floor X to floor Y, X different from Y;

Suppose the cost function is the sum over all steps in the solution of a number that is 1 for trips between adjacent floors, and 1.2 otherwise.

(a) Give a heuristic (nonnumeric reason) for choosing branches during search.

(b) Give a lower-bound evaluation function for use with A*.

(c) Would bidirectional search be a good idea for this and similar problems? Why?

(d) Approximate the size of the search space for T terminals and F floors.

(e) Draw the state graph after the first three states have had their successors found, in the solution of the given problem using A*. Use the evaluation function you gave, but not the heuristic. Don't allow returns to previous states. Draw the evaluation plus the cost of a state next to it. If ties arise, use a heuristic to choose, and say what heuristic you're using. Hint: use a compact notation for the state, so you don't have to write a lot.

9-13. (A) Consider this problem:

> A farmer wants to get a lion, a fox, a goose, and some corn across a river. There is a boat, but the farmer can only take one passenger in addition to himself on each trip, or else both the goose and the corn, or both the fox and the corn. The corn cannot be left with the goose because the goose will eat the corn; the fox cannot be left with the goose because the fox will eat the goose; and the lion cannot be left with the fox because the lion will

eat the fox. How does everything get across the river? Assume animals do not wander off when left alone.

(a) What is the search space?

(b) Give the starting and ending states.

(c) Give the operators.

(d) Draw the first two levels of the search graph. That's two besides the starting state.

(e) What is the average branching factor for these four levels? Disregard branches back to previous states.

(f) Give an upper bound on the size of the search space.

(g) Is this problem decomposable about an intermediate state?

9-14. Consider the two tasks of solving jigsaw puzzles and solving integration problems symbolically. For each, answer the following.

(a) What is the search space?

(b) What are the operators?

(c) What is the starting state?

(d) What are the final states?

(e) Is the task decomposable (breakable into subproblems that can be solved independently) about an intermediate state?

(f) Are the operators monotonic (applicable at any time, if applicable once)?

(g) Is one solution needed or the best solution?

(h) What is the initial branching factor?

(i) How (approximately) does the branching factor vary as the task proceeds?

9-15. (R,A) Consider the problem of designing electrical connections between places on the surface of a two-dimensional integrated circuit. Group the surface into square regions with a square grid. Consider this a search problem (a single search problem, not a group of search problems) in which there is only one operator: coat with metal the grid cell [X,Y] where X and Y are integers representing Cartesian coordinates. No grid cells have any metal coating in the starting state. In the goal state, electrical connections exist between each pair of a specified list of cell pairs; "electrical connection" means an unbroken sequence of adjacent coated grid cells linking a cell pair, while at the same time not linking them to any other cell pair. So for instance the goal might be to connect [10,20] to [25,29], and simultaneously [3,9] to [44,18], but not [10,20] to [3,9]. It's desired to find the coating pattern that uses the least amount of metal in achieving the goal connections.

(a) What is the search space?

(b) Is bidirectional search a good idea? Why?

(c) How does the branching factor vary as forward search proceeds?

(d) Give a heuristic for limiting forward search in this problem.

9-16. Consider the allocation of variables to registers done by an optimizing compiler for a programming language. Consider this as a search with just one operator: assign (allocate) occurrence N of variable V on line L of the program to be in register R. (Assume that variables must be in registers to use them in programs.) Since computers have a small fixed number of registers, and a poor allocation requires a lot of unnecessary instructions for moving data between registers and main memory, it's important to choose a good allocation if you want a compiler that generates the fastest possible code. For this problem, assume you do want to generate the fastest possible code. And assume this means the code with the fewest number of instructions, to make this simpler.

Notice that you can calculate speed for partial allocations of variables, not just complete allocations. That is, for all unbroken segments of code that mention register-assigned variables, you can count the number of instructions in the segments. Those numbers for those code segments can't change as new allocations of other variables are made, because one register allocation can't affect another.

(a) Which is the best search strategy?

(i) depth-first search
(ii) breadth-first search
(iii) best-first search
(iv) A* search

(b) Which is a heuristic for this problem? (only one answer is correct)

(i) "Count the number of machine instructions generated by a solution."
(ii) "Allocate registers to real-number variables first."
(iii) "Prefer forward chaining to backward chaining when there are few facts and many conclusions."
(iv) "Don't figure a register allocation of the same variable occurrence in a particular line twice."

(c) How does the branching factor change as the search proceeds (that is, with level in the search)? (choose only one answer)

(i) it decreases
(ii) it decreases or stays the same, depending on the state
(iii) it stays the same
(iv) it increases

9-17. Consider the task of proving a theorem from a set of postulates and assumptions. Suppose postulates, assumptions, and theorems are all either in the form of "A implies B" or just "A" where A and B are simple logical statements about the world. Suppose there is a way of taking any two postulates, assumptions, or previously derived theorems to get a conclusion, if any. For instance, if one statement says "A implies B" and another statement says "A", then we get a conclusion "B"; if one statement says "A implies B" and another statement says "B implies C", then we get a conclusion "A implies C".

(a) What is the search space?

(b) What is the branching factor, as a function of the number of steps taken?

(c) Give a search heuristic.

(d) Give a rough evaluation function for guiding search.

(e) Is the search monotonic?

(f) Is the search decomposable about an intermediate state?

(g) Is this a better problem for heuristics or for evaluation functions? Explain.

(h) Assuming we had a good approximate evaluation function, would the A* algorithm work well here? Explain.

(i) Would best-first search be a good idea here?

(j) Would bidirectional search be a good idea here?

9-18. Search has some surprising applications to numeric calculations. Consider the search for a numeric formula that approximates the value of some *target* variable from mathematical operations on a set of *study* variables. To test the accuracy of our approximation formula, we are given a set of *data points*, Prolog facts of the form

```
data_point([<study-variable-value-1>,<study-variable-value-2>, ... ],
  <target-variable-value>).
```

That is, the study variable values for a data point are stored in a list. Now think of this search problem as one of creating new list entries whose values match more closely the target variable values. To get these new study variables, we will do simple arithmetic operations on the old study variables. For example, take these data points:

```
data_point([1,2],6).
data_point([2,5],14).
data_point([3,2],10).
```

Then if we take the sum of the two study variables we can get a third study variable:

```
data_point([1,2,3],6).
data_point([2,5,7],14.)
data-point([3,2,5],10.)
```

and if we double those new values we can get

```
data_point([1,2,3,6],6).
data_point([2,5,7,14],14).
data_point([3,2,5,10],10).
```

and we have "explained" the value of the target variable as twice the sum of the original two study variables. In general, assume the following arithmetic operations are permissible on study variables:

> --multiplying by a constant or adding a constant to values of some study variable;
> --taking the negatives, squares, square roots, logarithms to the base 2 or powers of 2 of the values of some study variable;
> --taking the sum, difference, product, or quotient of the corresponding values for two different study variables

of the same data point for all data points.

(a) Considered as a search problem, what is the search space?

(b) What are the operators?

(c) What is the branching factor from any state?

(d) For the following starting data points, draw the first two levels (not counting the starting state as a level) of the search graph for the portion including only the "squaring" operation (that is, multiplying values by themselves).

```
data_point([1,2],1).
data_point([2,0],16).
data_point([3,1],81).
```

(e) Give an evaluation function for this problem (the problem in general, not just the data points in part (e)).

(f) If the variable values represent experimental measurements, it will be difficult for a formula to match the target variable exactly. What then would be a reasonable goal condition?

(g) Give a general-purpose heuristic (nonnumeric way) for choosing branches well for this general problem (for any set of data points, not just those in part (d)).

(h) Is bidirectional search a good idea for this problem? Why?

(i) Is A* search a good idea for this problem? Why?

(j) Explain why preconditions are necessary for some of the operators.

(k) Is the problem decomposable about an intermediate state? Why?

(l) Professional statisticians often do this sort of analysis. They claim they don't use search techniques very often. If so, how (in artificial intelligence terminology) must they be solving these problems?

9-19. (E) (a) Explain how playing chess or checkers requires search.

(b) How is this kind of search fundamentally different from those that we have considered in this chapter?

Go to book index

Implementing search

We have postponed consideration of search implementation to this chapter to better focus on the quite different issues involved. We will present programs for several kinds of search, working from a search problem described in an abstract way. We'll also use search as a springboard to discuss some advanced features of Prolog, including set-collection and "cut" predicates, and more of the wild and wacky world of backtracking.

Defining a simple search problem

First, we need a way to describe a search problem precisely. We ask the programmer to define the following predicates, the last two optional depending on the desired search strategy:

> --**successor(<old-state>,<desired new-state>)**: rules and facts defining this predicate give all possible immediate branches or state transitions. It's a function predicate with its first argument an input, a state, and second argument an output, a successor state. Both arguments are state descriptions. We emphasize that **<new-state>** must be an *immediate* successor.
>
> --**goalreached(<state>)**: rules and facts defining this predicate give the stopping conditions for the search. The argument is an input, a state. Multiple goal states are possible.
>
> --**eval(<state>,<evaluation>)**: rules and facts defining this predicate give the evaluation function. This is a function predicate taking an input state as first argument, and binding its second argument, an output, to an estimate of how close that state is to the nearest goal state.
>
> --**cost(<state-list>,<cost>)**: rules and facts defining this predicate give the cost function. This is a function predicate taking an input list of states as first argument, and binding its second argument, an output, to some nonnegative number representing the sum of the costs along the path through those states.

Of these four, the first is generally the hardest: it must define all the operators, incorporating their necessary conditions and describing precisely how they modify a state-description list. (The **successor** definitions may also include heuristics about operator use, though Section 10.11 will show a more general way to handle heuristics.) Successor definitions turn a search problem into a rule-based system, so we can steal ideas from Chapter 7.

Then to actually start searching, we ask that you query something like

```
?- search(<starting-state>,<answer-state-list>).
```

where the first argument is an input, the starting state (written in the format handled by the successor rules), and the second argument is an output, the discovered sequence of states, usually in reverse order (that is, from the goal state back to the start state).

As an example of this Prolog-style definition of a search problem, let's return to the example of Figure 9-1, redrawn as Figure 10-1. This is a city route planning problem in which our starting location is **a** and our goal location is **h**. States are the labeled intersections, so to define **successor** we must write a fact for every pair of intersections that connect directly:

```
successor(a,b).
successor(a,d).
successor(b,c).
successor(b,a).
successor(b,d).
successor(c,b).
successor(d,a).
successor(d,e).
successor(d,g).
successor(e,d).
successor(e,f).
successor(e,g).
successor(f,e).
successor(g,d).
successor(g,e).
successor(h,g).
```

For the **goalreached** condition we need:

```
goalreached(h).
```

And to start the program out, we will query a particular search predicate, like

```
?- depthsearch(a,Path).
```

to do depth-first search from start state **a**. (The search predicates we'll define in this chapter are **depthsearch**, **breadthsearch** (breadth-first), **bestsearch** (best-first), and **astarsearch** (A* search).) When search is done, the variable **Path** is bound to the list of intersections (states) we must travel through. We can use evaluation and cost functions to improve search if we like. A good evaluation function for this problem would be the straight-line distance to the goal (measuring on the map with a ruler), giving approximate distances for Figure 10-1 of:

```
eval(a,8).
eval(b,7).
eval(c,8).
eval(d,5).
eval(e,3).
eval(f,1).
eval(g,2).
eval(h,0).
```

The **cost** definition can be the actual distance along a road, which we can approximate from Figure 10-1 as

```
cost([X],0).
cost([X,Y|L],E) :- piece_cost(X,Y,E2), cost([Y|L],E3), E is E2 + E3.
```

```
piece_cost(a,b,3).
piece_cost(a,d,5).
piece_cost(b,c,1).
piece_cost(b,d,2).
piece_cost(d,e,2).
piece_cost(d,g,3).
piece_cost(e,f,2).
piece_cost(e,g,1).
piece_cost(g,h,2).
piece_cost(X,Y,C) :- piece_cost(Y,X,C).
```

A **cost** definition will often need recursion to handle paths of unknown length.

Defining a search problem with fact-list states

Now let's try representing a different kind of search problem, one more typical. The previous example was easy because states could be represented by letters. But often you can't name states in advance, as when you don't know what states are possible, and you must describe them by lists of facts.

Here's an example. This search concerns a very small part of car repair: putting nuts and washers onto bolts in the right way. Suppose we have two bolts. In the starting state (the top picture in Figure 10-2), nut **a**, washer **b**, and nut **c** are on the left bolt in that order, and nut **d**, washer **e**, and nut **f** are on the right bolt in that order. The goal is to get nut **c** on top of washer **e** (as for example in the bottom picture in Figure 10-2). To do this, we must work on one nut or washer at a time. We can remove one from the top of the nuts and washers on a bolt, place one on the top of the nuts and washers on a bolt, or just place one alone on the flat surface around the bolts. The bolts themselves can't be moved. The four nuts and two washers are the only parts available.

The only facts that change during the search are those about what parts (nuts or washers) rest on what other parts. Let's use **on(<part-1>,<part-2>,<bolt>)** facts for this, where **<part-1>** and **<part-2>** are codes for parts, and **<part-1>** is on the bolt named **<bolt>** directly above **<part-2>**. To keep track of parts *not* on the bolts, we'll represent them with **on** facts too, by **on(<part>,surface,none)**; parts can only rest on other parts when both are on bolts. To keep track of empty bolts, we'll use two additional permanent facts, **bolt(bolt1)** and **bolt(bolt2)**. So any state can be represented as a list of eight predicate expressions, six specifying the location of each of six parts, and two giving the names of the bolts. For instance, the starting state (the top picture in Figure 10-2) can be written

```
[on(a,b,bolt1),on(b,c,bolt1),on(c,surface,bolt1),on(d,e,bolt2),
  on(e,f,bolt2),on(f,surface,bolt2),bolt(bolt1),bolt(bolt2)]
```

In other words, like a Prolog database on its side. We could start a depth-first search by querying

```
?- depthsearch([on(a,b,bolt1),on(b,c,bolt1),on(c,surface,bolt1),
  on(d,e,bolt2),on(e,f,bolt2),on(f,surface,bolt2),
  bolt(bolt1),bolt(bolt2)],Answerpath).
```

The example final state in Figure 10-2 can be written

```
[on(a,surface,none),on(b,surface,none),on(c,e,bolt2),
  on(d,surface,none),on(e,f,bolt2),on(f,surface,bolt2),
  bolt(bolt1),bolt(bolt2)]
```

But let's say we're not so fussy about where every part ends up. Let's say we only want part **c** to be on part **e**. Then we can define the goal state by:

```
goalreached(S) :- member(on(c,e,bolt2),S).
```

That is, we can stop in state **S** whenever the facts true for **S** include **on(c,e,bolt2)**. The **member** predicate was defined in Section 5.5, and is true whenever some item is a member of some list:

```
member(X,[X|L]).
member(X,[Y|L]) :- member(X,L).
```

Successor rules like the preceding that work on fact lists usually just insert and delete facts from the list describing one state to get the list describing another. Typically, they only insert and delete a few. So to specify the effect of an operator, we need only list the fact *changes* made by an operator--everything else can be assumed constant. In problems with complicated state descriptions, it may be hard to figure which facts stay the same, because of "side effects". This is called the *frame problem* in artificial intelligence (not to be confused with the knowledge-partitioning "frames" we'll discuss in Chapter 12).

By referring to variables, just three successor rules are needed for the bolts problem, as shown in the following code. Their form is like the **append** and **delete** predicate definitions (Section 5.6). The first handles removing parts from bolts, and the second handles placing parts on bolts. (The intermediate predicate **cleartop** simplifies the rules.) The first successor rule says that if in some state the part **X** (1) is on a bolt and (2) doesn't have another part on it, then a successor state is one in which **X** is removed from the bolt and placed on the surface. Furthermore, the rule says that we can get the successor state from the old state by removing the old fact about where part **X** was (with the **delete** predicate), and adding a new fact that **X** is alone on the surface (with the stuff in brackets on the rule left side, the last thing done before the rule succeeds). The second successor rule says that if in some state the part **X** (1) doesn't have another part on it, and (2) another part **Z** doesn't have anything on it either, and (3) **Z** is on a bolt, and (4) **Z** is different from **X**, then a possible successor state has **X** placed on **Z**. And we can get a description of the new state by removing the old location of **X** and adding the new location. The third successor rule says that if a bolt is empty in some state, then we can put on it any part with a clear top.

```
successor(S,[on(X,surface,none)|S2]) :- member(on(X,Y,B),S),
  not(B=none), cleartop(X,S), delete(on(X,Y,B),S,S2).
successor(S,[on(X,Z,B2)|S2]) :- member(on(X,Y,B),S),
  cleartop(X,S), member(on(Z,W,B2),S), not(B2=none),
  cleartop(Z,S), not(X=Z), delete(on(X,Y,B),S,S2).
successor(S,[on(X,surface,B2)|S2]) :- member(on(X,Y,B),S),
  cleartop(X,S), member(bolt(B2),S), not(member(on(Z,W,B2),S)),
  delete(on(X,Y,B),S,S2).


cleartop(Part,State) :- not(member(on(X,Part,B),State)).


delete(X,[X|L],L).
delete(X,[X|L],L2) :- delete(X,L,L2).
delete(X,[Y|L],[Y|L2]) :- not(X=Y), delete(X,L,L2).
```

In the preceding, **cleartop** just checks that there's nothing resting on a part. The **delete** predicate comes from Section 5.6: it removes all occurrences of its first argument (an input) from the list that is its second argument (an input), binding the resulting list to the third argument (the output).

Implementing depth-first search

Prolog interpreters work depth-first, so it isn't hard to implement a general depth-first search facility in Prolog. As with the programs in Chapter 7, we'll divide code for a search into two files: a problem-dependent or "problem-defining" file containing, **successor**, **goalreached**, **eval**, and **cost** definitions discussed in Section 10.1, and a problem-independent file containing search machinery. Here is the problem-independent depth-first-search file in its entirety | REFERENCE 1|: .FS | REFERENCE 1| When using all the programs in this chapter, be careful not to redefine or duplicate definitions of the predicate names used here, or you can get into serious trouble. That particularly applies to duplication of the classic list-predicate definitions **member**, **length**, and **append**. .FE

```
/* Problem-independent code for depth-first search */
depthsearch(Start,Ans) :- depthsearch2(Start,[Start],Ans).


depthsearch2(State,Statelist,Statelist) :- goalreached(State).
depthsearch2(State,Statelist,Ans) :- successor(State,Newstate),
 not(member(Newstate,Statelist)),
 depthsearch2(Newstate,[Newstate|Statelist],Ans).


member(X,[X|L]).
member(X,[Y|L]) :- member(X,L).
```

Predicate **depthsearch** is the top level. Its first argument is an input, bound to a description of the starting state, and its second argument is the output, bound when search is done to the solution path in reverse order. The **depthsearch** rule just initializes a third argument (the middle one) for the predicate **depthsearch2**.

The recursive **depthsearch2** predicate does the real work of the program. Let's first show it declaratively; the next section will explain it mostly procedurally. Its three arguments are the current state (an input), the path followed to this state (an input), and the eventual path list found (an output). The first **depthsearch2** rule says that if the current state is a goal state, bind the output (third argument) to the second argument, the list of states we went through to get here. Otherwise, the second rule says to find some successor of the current state not previously encountered on the path here (that is, avoid infinite loops), put this successor on the front of the path list, and recursively search for the goal from the new state. The **member** predicate is from the last section.

The key to this program is the **successor** predicate expression in the second **depthsearch2** rule. Backing up in a search problem means backtracking to that expression. Whenever a state has no successors, or all its successors have been tried and found not to lead to a goal, the second **depthsearch2** rule fails. Since it's the *last* **depthsearch2** rule, the program returns to where it was called--or it "backs up". If the call of **depthsearch2** was (as usually) a recursive one from the same second **depthsearch2** rule at the next highest level of recursion, backtracking goes to the **not** (which like all **not**s, can never succeed on backtracking), and then immediately to the **successor** predicate. If another successor can be found for this earlier state, it is then taken. Otherwise, *this* invocation of the rule also fails, and backing up and backtracking happens again.

A depth-first example

Let's simulate the depth-first program on an example, to illustrate its procedural interpretation. The previous city-route example is a little too complicated for a good show, so let's try the following. Suppose in some city (see Figure 10-3) that two-way streets connect intersection **a** with intersection **b**, intersection **b** to intersection **d**, and intersection **d** to intersection **e**. Suppose also that one-way streets connect intersection **b** to intersection **c**, and intersection **a** to intersection **d**. Then the following **successor** facts hold; assume they're put in this order.

```
successor(a,b).
successor(b,a).
successor(b,c).
successor(a,d).
successor(b,d).
successor(d,b).
successor(d,e).
successor(e,d).
```

Assume for this problem that the starting state is **a**, and there is only one goal state **e**. Then the problem-definition file must also include

```
goalreached(e).
```

Now let's follow what happens when we query

```
?- depthsearch(a,Answer).
```

and the problem-independent file of the last section is loaded in addition to the previous problem-dependent specification. The action is summarized in Figure 10-4.

1. The predicate **depthsearch** is called with its first argument bound to **a** and its second argument unbound. So predicate **depthsearch2** is called with first argument **a**, second argument **[a]**, and third argument unbound.

2. The goal is not reached in state **a**, so the second rule for **depthsearch2** is tried.

3. In the **successor** facts, the first successor of state **a** is **b**, and **b** is not a member of the list of previous states, **[a]**. So **depthsearch2** is recursively called with first argument **b**, second argument **[b,a]**, and third argument unbound.

4. For this second level of **depthsearch2** recursion, the state **b** is not a goal state; the first successor listed for **b** is **c**, and **c** isn't in the list of previous states, **[b,a]**. So we recursively call **depthsearch2**, now with first argument **c**, second argument **[c,b,a]**, and third argument unbound.

5. For this third level of recursion, new state **c** is not a goal state nor does it have successors, so both

rules for **depthsearch2** fail. We must backtrack, "backing up" to the previous state **b** at the second level, and hence to the recursive **depthsearch2** call in the last line of the last **depthsearch2** rule.

6. The **not** fails on backtracking, as all **not**s do, so we backtrack further to the **successor** predicate in the second rule for **depthsearch2**, which chooses successors. Backtracking to here means we want a different successor for state **b** than **c**. And the only other successor of **b** is **d**. So we resume forward progress through the second **depthsearch2** rule with Newstate bound to **d**. This **d** is not in the list of previously visited states **[b,a]**, so we recursively call **depthsearch2** with first argument **d**, second argument **[d,b,a]** (**c** was removed in backtracking), and third argument unbound.

7. For this new third-level call, the new state **d** is not a goal, so we find a successor of it. Its first-listed successor is **b**, but **b** is a member of the list of previous states **[d,b,a]**, so we backtrack to find another successor.

8. The only other successor fact for **d** mentions state **e**. This isn't a member of **[d,b,a]**, so we recursively call **depthsearch2** with the **e** as first argument, **[e,d,b,a]** as second argument, and an unbound variable (still) as third argument.

9. But for this fourth level of recursion, state **e** is a goal state, and the **goalreached** predicate succeeds. So the first rule for **depthsearch2** succeeds, binding the third argument **Statelist** (finally!) to the list of states visited on the path here in reverse order, **[e,d,b,a]**. Now all other levels of **depthsearch2** recursion succeed because the recursive call was the last predicate expression in the rule (in other words, we always did tail recursion).

l0. So query variable **Answer** is bound to **[e,d,b,a]**.

Notice this is not the shortest solution to the problem, as is common with depth-first search.

Implementing breadth-first search

We can write code for problem-independent breadth-first search, to load as an alternative to the depth-first problem-independent code.

Recall from the last chapter that breadth-first search finds states level-by-level (that is, by distance in number of branches from the starting state). To do this, it must keep facts about all states found but whose successors haven't yet been found. Those states are an *agenda*; each represents further work to do. One simple way to implement breadth-first search is to make the agenda a queue (see Appendix C), a data structure for which the first thing added is always the first thing removed. We begin with a queue consisting of just the starting state, and anytime we find successors, we put them on the end of the queue. That way we are guaranteed to not try (find successors of) any states at level |N| until all states at level |N-1| have been tried.

We can implement agendas in Prolog, by facts with predicate name **agenda**, one for each unexplored (successors-not-found) state. To get the effect of a queue, we can put new states at the end of the agenda by the built-in predicate **assertz** (introduced in Section 6.1), so the first fact will always be the one that has been on the agenda the longest, and the first we'll find when we query the agenda facts. Predicate **agenda** can have two arguments: a state, and the path followed to reach it. As with depth-first search, the second argument is needed because (1) checking states against it prevents some of the possible infinite loops, and (2) its value for the goal state is the answer to the search problem.

We also should keep **oldagenda** facts. These, with the same two arguments as **agenda** facts, can record "exhausted" states, states for which we have found all successors. Checking against **oldagenda** facts before adding a new **agenda** fact prevents other infinite loops. This checking also ensures that any path we find to a state **S** has the fewest number of branches of any path to **S**, because it was found first and breadth-first works level by level.

Here is the problem-independent breadth-first search code | REFERENCE 2|, whose predicate hierarchy is given in Figure 10-5. (Those strange exclamation points "**!**" will be explained in Section 10.7, and the **bagof** in Section 10.6.) .FS | REFERENCE 2| This won't work for some Prolog dialects, those that can't handle dynamic additions of new backtracking points with **assertz**. For such implementations, we can get breadth-first search from the best-first search program **bestsearch** given later in this chapter, by including two extra lines with it:

eval(S,T) :- time(T), retract(time(T)), T2 is T+1, asserta(time(T2)).
eval(S,0) :- not(time(T)), asserta(time(1)). .FE

```
/* Problem-independent breadth-first search */
breadthsearch(Start,Ans) :- cleandatabase,
  asserta(agenda(Start,[Start])), agenda(State,Oldstates),
  find_successors(State,Oldstates,Newstate),
  goalreached(Newstate), agenda(Newstate,Ans),
  retract(agenda(Newstate,Ans)),
  asserta(oldagenda(Newstate,Ans)), measurework.


find_successors(State,Oldstates,Newstate) :-
  successor(State,Newstate), not(State = Newstate),
  not(agenda(Newstate,S)), not(oldagenda(Newstate,S)),
  assertz(agenda(Newstate,[Newstate|Oldstates])).
find_successors(State,Oldstates,Newstate) :-
  retract(agenda(State,Oldstates)),
  asserta(oldagenda(State,Oldstates)), fail.


cleandatabase :- abolish(oldagenda,2), abolish(agenda,2), !.
cleandatabase :- abolish(agenda,2), !.
cleandatabase.


measurework :- bagof([X,Y],agenda(X,Y),Aset), length(Aset,Len),
 bagof([X2,Y2],oldagenda(X2,Y2),A2set), length(A2set,Len2),
 write(Len), write(' incompletely examined state(s) and '),
 write(Len2),write(' examined state(s).'),!.
measurework :- bagof([X,Y],oldagenda(X,Y),Aset),
 length(Aset,Len), write('no incompletely examined states and '),
 write(Len), write(' examined state(s).'),!.


length([],0).
length([X|L],N) :- length(L,N2), N is N2+1.
```

The predicate **breadthsearch** starts by removing any **agenda** and **oldagenda** facts remaining from previous searches. It "seeds" or starts the agenda with a single item, the starting state. It then spends most of its time bouncing back and forth between the next three predicate expressions **agenda**, **find_successors**, and **goalreached**. The **agenda** retrieves an agenda state, **find_successors** finds a successor of it (as explained in a moment), and **goalreached** checks whether it's done. Most of the time it won't be. So most of the time it backtracks to the **find_successors** call to find another successor, or if there aren't any more, it backtracks to the **agenda** call to pick the next state on the agenda. When a goal state is found, it cleans up the agenda, and binds the answer variable (the second argument to **breadthsearch** as with **depthsearch**) to the path used to get there. Finally, it prints the size of the agenda and the oldagenda to show how much work it did.

The two **find_successors** rules are the key to the program. Function predicate **find_successors** has three arguments: the current state (an input), the path there (an input), and a successor state (an output). The right side of the first **find_successors** rule calls first on the **successor** definition, just as in the depth-first search program. If a successor is found, it is checked to be (1) not the current state, (2) not on the agenda, and (3) not on the oldagenda. Only if these tests succeed is the new successor added to the agenda. The first **find_successors** rule is repeatedly backtracked into to generate all the successors of a state; this backtracking is forced by the usually-failing **goalreached** expression in **breadthsearch**.

The second **find_successors** rule applies whenever the first fails, or whenever no further successors can be found for a state. It removes the "exhausted" state from the agenda and adds it to the oldagenda. Then by a **fail**, it forces the top-level predicate **breadthsearch** to pick a new state to find successors of. As we explained before, the next state picked will always be the oldest remaining agenda fact because of the **assertz**.

If the agenda ever becomes empty (that is, there are no new states to be found), then the **agenda** in **breadthsearch** fails, and then the **asserta** fails (there's never a new way to assert something), and then the **cleandatabase** fails (though we can't explain why this last for several pages yet). So the interpreter would type **no**.

As an example, here is the result of running the breadth-first program on the bolts problem defined in Section 10.2. Three solution paths were found, differing only in when part **e** is removed relative to the removals of parts **a** and **b**. (Carriage returns have been added to improve readability.)

```
?- breadthsearch([on(a,b,bolt1),on(b,c,bolt1),on(c,surface,bolt1),
  on(d,e,bolt2),on(e,f,bolt2),on(f,surface,bolt2)],Answer).
81 incompletely examined state(s) and 1 examined state(s).
Answer=
[[on(c,e,bolt2),on(d,surface,none),on(b,surface,none),
  on(a,surface,none),on(e,f,bolt2),on(f,surface,bolt2)],
[on(d,surface,none),on(b,surface,none),on(a,surface,none),
  on(c,surface,bolt1),on(e,f,bolt2),on(f,surface,bolt2)],
[on(b,surface,none),on(a,surface,none),on(c,surface,bolt1),
  on(d,e,bolt2),on(e,f,bolt2),on(f,surface,bolt2)],
[on(a,surface,none),on(b,c,bolt1),on(c,surface,bolt1),
  on(d,e,bolt2),on(e,f,bolt2),on(f,surface,bolt2)],
[on(a,b,bolt1),on(b,c,bolt1),on(c,surface,bolt1),
  on(d,e,bolt2),on(e,f,bolt2),on(f,surface,bolt2)]] ;

102 incompletely examined state(s) and 2 examined state(s).
Answer=
[[on(c,e,bolt2),on(b,surface,none),on(d,surface,none),
  on(a,surface,none),on(e,f,bolt2),on(f,surface,bolt2)],
[on(b,surface,none),on(d,surface,none),on(a,surface,none),
  on(c,surface,bolt1),on(e,f,bolt2),on(f,surface,bolt2)],
[on(d,surface,none),on(a,surface,none),on(b,c,bolt1),
  on(c,surface,bolt1),on(e,f,bolt2),on(f,surface,bolt2)],
[on(a,surface,none),on(b,c,bolt1),on(c,surface,bolt1),
  on(d,e,bolt2),on(e,f,bolt2),on(f,surface,bolt2)],
[on(a,b,bolt1),on(b,c,bolt1),on(c,surface,bolt1),
  on(d,e,bolt2),on(e,f,bolt2),on(f,surface,bolt2)]] ;

152 incompletely examined state(s) and 3 examined state(s).
Answer=
[[on(c,e,bolt2),on(b,surface,none),on(a,surface,none),
  on(d,surface,none),on(e,f,bolt2),on(f,surface,bolt2)],
[on(b,surface,none),on(a,surface,none),on(d,surface,none),
  on(c,surface,bolt1),on(e,f,bolt2),on(f,surface,bolt2)],
[on(a,surface,none),on(d,surface,none),on(b,c,bolt1),
  on(c,surface,bolt1),on(e,f,bolt2),on(f,surface,bolt2)],
[on(d,surface,none),on(a,b,bolt1),on(b,c,bolt1),
  on(c,surface,bolt1),on(e,f,bolt2),on(f,surface,bolt2)],
[on(a,b,bolt1),on(b,c,bolt1),on(c,surface,bolt1),
  on(d,e,bolt2),on(e,f,bolt2),on(f,surface,bolt2)]]
```

Collecting all items that satisfy a predicate expression

A feature of the **breadthsearch** program we haven't explained is the **bagof** predicate in the **measurework** rules. This predicate we used before in Chapter 8 to implement "or-combination", and it is built-in in most Prolog dialects (though easily defined in Prolog). It collects into a list all the values for some variable that satisfy a predicate expression, much like the **forall** (defined in Section 7.12) in reverse. Predicate **bagof** takes three arguments: an input variable, an input predicate expression containing that variable, and an output list to hold all possible bindings of that variable which satisfy that expression. (Some variants of **bagof** delete duplicates in the result.)

Here's an example. Suppose we have this database:

```
boss_of(mary,tom).
boss_of(mary,dick).
boss_of(dick,harry).
boss_of(dick,ann).
```

Suppose we want a list of all people that Mary is the boss of. We can type

```
?- bagof(X,boss_of(mary,X),L)
```

and the interpreter will type

```
L=[tom,dick]
```

and **X** won't be printed because it's just a placeholder.

We can put, within the expression that is the second argument to **bagof**, variables besides the one we are collecting; the interpreter will try to bind them too. So if, for the preceding database, we type

```
?- bagof(X,boss_of(Y,X),L).
```

the interpreter will type

```
Y=mary, L=[tom,dick]
```

for its first answer. If we then type a semicolon, it will type

```
Y=dick, L=[harry,ann]
```

for its second answer.

The first argument to **bagof** can be a list. That is, we can search for a set of values satisfying a predicate instead of just one value. For instance, we can query

```
?- bagof([X,Y],boss_of(X,Y),L).
```

and receive the answer

```
L=[[mary,tom],[mary,dick],[dick,harry],[dick,ann]]
```

This query form is used in the **measurework** rules in the **breadthsearch** program.

The **bagof** predicate can be defined this way (provided you have no other predicate named **zzz** in your program):

```
bagof(X,P,L) :- asserta(zzz([])), fail.
bagof(X,P,L) :- call(P), zzz(M), retract(zzz(M)),
  asserta(zzz([X|M])), fail.
bagof(X,P,L) :- zzz(L), retract(zzz(L)).
```

The cut predicate

We still must explain those strange exclamation points (the "**!**") in the **breadthsearch** program. These are a special built-in predicate of no arguments called the *cut*, whose meaning is exclusively procedural. The cut always succeeds when the interpreter first encounters it, but has a special side effect: it prevents backtracking to it by throwing away the necessary bookkeeping information. This can improve the efficiency of Prolog programs, but it is also necessary to make some programs work properly, those for which backtracking just doesn't make sense.

Usually the cut predicate expression is last in a rule, as in **cleandatabase** and the two **measurework** rules in the **breadthsearch** program. It can be paraphrased as: "Don't ever backtrack into this rule. What's more, don't ever try another rule to satisfy the goal that this rule tried to satisfy. That goal is dead. So fail it." (Note that a cut has no effect on the next call of the rule, even a call with the same arguments as before: the prohibition of backtracking just applies to the call in which the cut was encountered.) So a cut symbol forces behavior like that of a subprocedure in a conventional programming language, in that once a subprocedure is done it can only be reentered by a different call--except that a Prolog "subprocedure" is all the rules with the same left-side predicate name, not just a single rule. So a cut at the end of a rule means you want only one solution to the query of that rule. This often is true for "existential quantification" queries in which we check existence of something of a certain type, and we don't care what. For instance, the **member** predicate from Section 5.5

```
member(X,[X|L]).
member(X,[Y|L]) :- member(X,L).
```

is often used this way, when it is queried with both arguments bound. For instance:

```
?- member(b,[a,b,c,b,e,a,f).
```

Recursion will find the first **b** in the list and the interpreter will type **yes**. If we were to type a semicolon, recursion would find the second occurrence of **b** and the interpreter would type **yes** again. But finding that second occurrence doesn't make sense in most applications; the answer to the query must be the same, after all. A backtracking **member** is useless in our depth-first and breadth-first search programs, where **member** is enclosed in a **not**, since the Prolog interpreter doesn't backtrack into **not**s. And backtracking into **member** is unnecessary whenever there can't be duplicates in a list. (But we do need a backtracking **member** in the bolts example of Section 10.2, to choose alternative parts to move by querying **member** with an unbound first argument.) A non-backtracking **member** can be obtained by just inserting a cut symbol in the backtracking **member**:

```
singlemember(X,[X|L]) :- !.
singlemember(X,[Y|L]) :- singlemember(X,L).
```

We don't need a cut symbol at the end of the second rule, because when it fails no more possibilities remain.

We don't actually need a cut symbol to define **singlemember**, for we could say equivalently:

```
singlemember(X,[X|L]).
singlemember(X,[Y|L]) :- not(X=Y), singlemember(X,L).
```

But this is slower because the extra expression **not(X=Y)** must be tested on every recursion; the cut predicate expression is done at most once for any query.

A related use of the cut predicate is to do something only once instead of repeatedly. For instance, here's the predicate from Section 5.6 that deletes all occurrences of an item **X** from a list **L**:

```
delete(X,[],[]).
delete(X,[X|L],M) :- delete(X,L,M).
delete(X,[Y|L],[Y|M]) :- not(X=Y), delete(X,L,M).
```

Suppose we want a predicate that deletes only the first item **X** from list **L**. We can remove the recursion in the second rule, remove the **not(X=Y)** in the third rule, and insert a cut symbol:

```
deleteone(X,[],[]).
deleteone(X,[X|L],L) :-  !.
deleteone(X,[Y|L],[Y|M]) :- deleteone(X,L,M).
```

The cut symbol is important here, because if we just omit it like this:

```
deleteone(X,[],[]).
deleteone(X,[X|L],L).
deleteone(X,[Y|L],[Y|M]) :- deleteone(X,L,M).
```

then **deleteone** will give a correct first answer, but wrong subsequent answers on backtracking, just like the similar mistake-making **delete** discussed in Section 5.6.

The cut predicate can also be used merely to improve efficiency. Recall the definition of the maximum of a list in Section 5.5:

```
max([X],X).
max([X|L],X) :- max(L,M), X>M.
max([X|L],M) :- max(L,M), not(X>M).
```

When the first two rules fail, the computation of the maximum of a list is done twice: once in the second rule, and once in the third rule. This is wasteful. So we can define it:

```
max([X],X) :- !.
max([X|L],M) :- max(L,M), not(X>M), !.
max([X|L],X).
```

Here we've changed the order of the last two rules and removed the redundant **max** call. The cuts guarantee that if we ever backtrack into **max**, we won't start taking the third rule when the second rule was taken before, and thereby get wrong answers.

A cut predicate can be put in the middle of a rule. Then it means that backtracking is allowed to the right of it, but that if the interpreter ever tries to backtrack to its left, both the rule and the goal that invoked it will fail unconditionally.

Nothing in life is free, so it's not surprising that the efficiency advantages of the cut predicate have the associated disadvantage of restricting multiway use (see Section 3.3) of predicate definitions. That's because the cut is a purely procedural feature of Prolog, with no declarative meaning. But if you're sure you'll only query a definition with one particular pattern of bindings, multiway use is not an issue, and cut predicates can be used freely to improve efficiency.

Iteration with the cut predicate (*)

The cut predicate provides a way to write iterative Prolog programs in a way more general than the **forall** and **doall** of Section 7.12 and **bagof** of Section 10.6. It gives a sometimes better way to repeat things than by backtracking, since backtracking doe things in reverse order, and that can be confusing or even wrong.

To get a true "do-until", or in other words to repeatedly query some predicate expression Pred until some other predicate expression **done** holds, query predicate **iterate** with argument Pred:

```
iterate(Pred) :- repeat, iterate2(Pred), done.
iterate2(Pred) :- call(Pred), !.
repeat.
repeat :- repeat.
```

Here **repeat** is a predicate that always succeeds, and what's more, always succeeds on backtracking (unlike **1=1** which always succeeds once, but fails on backtracking because it can't succeed in a new way). The **repeat** is built-in in many Prolog dialects, but it's easy to define as you see.

Predicate **iterate** will hand the expression Pred to **iterate2** for execution using the **call** predicate explained in Section 7.12. Then **iterate** checks the **done** condition, which usually fails. At this point, the cut in **iterate** is important, because it prevents backtracking into **iterate2** and Pred. So **iterate2** fails, and the interpreter returns to the **repeat**. But **repeat** always succeeds anew on backtracking (it just recurses once more as a new way to succeed), and so the interpreter returns to **iterate2**, and Pred is executed again *in the forward direction*. So the cut predicate forces the interpreter to execute Pred like within a loop in a conventional programming language: always forward. (Note that Pred can contain arguments.)

One disadvantage of the preceding is that **iterate** can never fail. This would cause an infinite loop if **done** has a bug preventing it from ever succeeding. So we might distinguish **donegood** and **donebad** conditions, both user-defined, for when the iteration should stop with success and failure respectively:

```
iterate(Pred) :- repeatcond, iterate2(Pred), donegood.
iterate2(Pred) :- call(Pred), !.
repeatcond.
repeatcond :- not(donebad), repeatcond.
```

Another kind of iteration increases a counter at each iteration, like the "FOR" construct in Pascal and the "DO" construct in Fortran which iterate for K=1 to N:

```
foriterate(Pred,N) :- asserta(counter(0)), repeat, counter(K),
  K2 is K+1, retract(counter(K)), asserta(counter(K2)),
  iterate2(Pred), K=<N.
iterate2(Pred) :- call(Pred), !.
repeat.
repeat :- repeat.
```

To access the counter at any time inside the rules invoked by calling Pred, you query predicate **counter**.

Implementing best-first search (*)

Now we can show our best-first search program. To use it you need (besides **successor** and **goalreached** definitions) a definition of a function predicate **eval** of two arguments. As we said in Section 10.1, the first argument to **eval** is an input state, and its second is an output number, a nonnegative evaluation of that state.

The best-first program keeps an agenda of states like the breadth-first program, but each agenda fact has a third argument holding the evaluation function value for the state. (It makes sense to compute this when we find the state and put it on the agenda, so we only do it once per state.) And when we select a state from an agenda with the

pick_best_state predicate, we must take the state with the minimum evaluation function value, not just the first one on the agenda. Our best-first search program also has iteration in several places where the breadth-first search program used recursion, so it's a more efficient program. Here's the best-first search program, whose predicate hierarchy appears in Figure 10-6:

```
/* Problem-independent best-first search */
bestsearch(Start,Goalpathlist) :- cleandatabase,
  add_state(Start,[]), repeatifagenda,
  pick_best_state(State,Pathlist),
  add_successors(State,Pathlist), agenda(State,Goalpathlist,E),
  retract(agenda(State,Goalpathlist,E)), measurework.


pick_best_state(State,Pathlist) :-
  asserta(beststate(dummy,dummy,dummy)),
  agenda(S,SL,E), beststate(S2,SL2,E2), special_less_than(E,E2),
  retract(beststate(S2,SL2,E2)), asserta(beststate(S,SL,E)), fail.
pick_best_state(State,Pathlist) :- beststate(State,Pathlist,E),
  retract(beststate(State,Pathlist,E)), not(E=dummy), !.


add_successors(State,Pathlist) :- goalreached(State), !.
add_successors(State,Pathlist) :- successor(State,Newstate),
  add_state(Newstate,Pathlist), fail.
add_successors(State,Pathlist) :-
  retract(agenda(State,Pathlist,E)),
  asserta(usedstate(State)), fail.


add_state(Newstate,Pathlist) :- not(usedstate(Newstate)),
  not(agenda(Newstate,P,E)), eval(Newstate,Enew),
  asserta(agenda(Newstate,[Newstate|Pathlist],Enew)), !.
add_state(Newstate,Pathlist) :- not(eval(Newstate,Enew)),
  write('Warning: your evaluation function failed on state '),
  write(Newstate), nl, !.

/* Utility functions */
repeatifagenda.
repeatifagenda :- agenda(X,Y,Z), repeatifagenda.


special_less_than(X,dummy) :- !.
special_less_than(X,Y) :- X<Y.


cleandatabase :- checkabolish(agenda,3), checkabolish(usedstate,1),
  checkabolish(beststate,1), checkabolish(counter,1).
checkabolish(P,N) :- abolish(P,N), !.
checkabolish(P,N).



measurework :- countup(agenda(X,Y,Z),NA), countup(usedstate(S),NB),
  write(NA), write(' incompletely examined state(s) and '),
  write(NB),write(' examined state(s)'), !.



countup(P,N) :- asserta(counter(0)), call(P), counter(K),
  retract(counter(K)), K2 is K+1, asserta(counter(K2)), fail.
countup(P,N) :- counter(N), retract(counter(N)), !.
```

The top-level **bestsearch** predicate iterates by repeatedly picking a state from the agenda. It initializes and cleans up around a kernel of commands repeatedly invoked. But the iteration is done differently from the **breadthsearch** program, with the **bestsearch** bouncing among a **repeatifagenda** predicate on the left, the **pick_best_state** predicate, and an **add_successors** on the right. The **repeatifagenda** is an instance of the **repeatcond** discussed in the last section.

The **pick_best_state** chooses the minimum-evaluation state by iterating over the agenda states; the cut predicate at the end of its definition ensures that it finds only one such state per call. The **add_successors** checks whether some state **S** is a goal state (then succeeding), and otherwise adds all the acceptable successors of **S** to the agenda (then failing); the predicate **add_state** runs the necessary acceptance checks on a new state before adding it to the agenda. When **add_successors** finally succeeds, the path associated with its state argument must be the solution, so this is retrieved and bound to the answer **Goalpathlist**.

As an example, let's use best-first search on the bolts problem of Section 10.2. We need an evaluation function. Our goal is to get part **c** on part **e**, so we could take the sum of the number of parts on top of both **c** and **e**. That is, we could measure the degree of "burial" of each. But then many states have evaluation zero. Instead, let's try:

```
eval(S,0) :- goalreached(S).
eval(S,N) :- burial(c,S,N1), burial(e,S,N2), N is N1+N2+1.



burial(P,S,0) :- cleartop(P,S).
burial(P,S,N) :- member(on(X,P,B),S), burial(X,S,N2), N is N2+1.
```

Running **bestsearch** on the same starting state as before, we get the third answer for breadth-first search:

```
?- bestsearch([on(a,b,bolt1),on(b,c,bolt1),on(c,surface,bolt1),
  on(d,e,bolt2),on(e,f,bolt2),on(f,surface,bolt2)],A).
23 incompletely examined state(s) and 4 examined state(s)
A=[[on(c,e,bolt2),on(b,surface,none),on(a,surface,none),
  on(d,surface,none),on(e,f,bolt2),on(f,surface,bolt2)],
[on(b,surface,none),on(a,surface,none),on(d,surface,none),
  on(c,surface,bolt1),on(e,f,bolt2),on(f,surface,bolt2)],
[on(a,surface,none),on(d,surface,none),on(b,c,bolt1),
  on(c,surface,bolt1),on(e,f,bolt2),on(f,surface,bolt2)],
[on(d,surface,none),on(a,b,bolt1),on(b,c,bolt1),
  on(c,surface,bolt1),on(e,f,bolt2),on(f,surface,bolt2)],
[on(a,b,bolt1),on(b,c,bolt1),on(c,surface,bolt1),
  on(d,e,bolt2),on(e,f,bolt2),on(f,surface,bolt2)]]
```

But look how many fewer states were found--27 (23+4) versus 82. That's more efficient search.

Implementing A* search (*)

A* search is like best-first except we must add path cost to the evaluation function. So as we said, the user must define a **cost** function predicate with two arguments, an input path and an output holding the computed cost of that path.

Our A* program is intended to be forgiving. So it still works if you don't give it a lower-bound cost function, though you may need to type semicolons to get the right (lowest-cost) answer. Also without a lower-bound evaluation function, the first path found to any state **S** may not be lowest-cost, so if we find a lower-cost path to **S** later, we must revise the path lists of everything on the agenda mentioning **S**.

Here's the program (whose predicate hierarchy is summarized in Figure 10-7):

```
/* Problem-independent A* search code. */
/* Note: "cost" must be nonnegative.  The "eval" should be a lower */
/* bound on cost in order for the first answer found to be */
/* guaranteed optimal, but the right answer will be reached */
/* eventually otherwise.  */


astarsearch(Start,Goalpathlist) :- cleandatabase,
  add_state(Start,[]), repeatifagenda,
  pick_best_state(State,Pathlist),
  add_successors(State,Pathlist), agenda(State,Goalpathlist,C,D),
  retract(agenda(State,Goalpathlist,C,D)), measurework.


pick_best_state(State,Pathlist) :-
  asserta(beststate(dummy,dummy,dummy)),
  agenda(S,SL,C,D), beststate(S2,SL2,D2), special_less_than(D,D2),
  retract(beststate(S2,SL2,D2)), asserta(beststate(S,SL,D)), fail.
pick_best_state(State,Pathlist) :- beststate(State,Pathlist,D),
  retract(beststate(State,Pathlist,D)), not(D=dummy), !.


add_successors(State,Pathlist) :- goalreached(State), !.
add_successors(State,Pathlist) :- successor(State,Newstate),
  add_state(Newstate,Pathlist), fail.
add_successors(State,Pathlist) :-
  retract(agenda(State,Pathlist,C,D)),
  asserta(usedstate(State,C)), fail.

/* If you are sure that your evaluation function is always
/* a lower bound on the cost function, then you can delete the */
/* first rule for "agenda_check" and the first rule for */
/* "usedstate_check", and delete the entire definitions of */
/* "fix_agenda", "replace_front", and "append". */
add_state(Newstate,Pathlist) :- cost([Newstate|Pathlist],Cnew), !,
  agenda_check(Newstate,Cnew), !,
  usedstate_check(Newstate,Pathlist,Cnew), !,
  eval(Newstate,Enew), D is Enew + Cnew,
  asserta(agenda(Newstate,[Newstate|Pathlist],Cnew,D)), !.
add_state(Newstate,Pathlist) :-
  not(cost([Newstate|Pathlist],Cnew)),
  write('Warning: your cost function failed on path list '),
  write(Pathlist), nl, !.
add_state(Newstate,Pathlist) :- not(eval(Newstate,Enew)),
  write('Warning: your evaluation function failed on state '),
  write(Newstate), nl, !.


agenda_check(S,C) :- agenda(S,P2,C2,D2), C<C2,
  retract(agenda(S,P2,C2,D2)), !.
agenda_check(S,C) :- agenda(S,P2,C2,D2), !, fail.
agenda_check(S,C).


usedstate_check(S,P,C) :- usedstate(S,C2), C<C2,
  retract(usedstate(S,C2)), asserta(usedstate(S,C)), !,
  fix_agenda(S,P,C,C2).
usedstate_check(S,P,C) :- usedstate(S,C2), !, fail.
usedstate_check(S,P,C).


fix_agenda(S,P,C,OldC) :- agenda(S2,P2,C2,D2),
  replace_front(P,S,P2,Pnew), cost(Pnew,Cnew),
  Dnew is D2+C-OldC, retract(agenda(S2,P2,C2,D2)),
  asserta(agenda(S2,Pnew,Cnew,Dnew)), fail.


replace_front(P,S,P2,Pnew) :- append(P3,[S|P4],P2),
  append(P,[S|P4],Pnew), !.

/* Utility functions */
repeatifagenda.
repeatifagenda :- agenda(X,Y,Z,W), repeatifagenda.


special_less_than(X,dummy) :- !.
special_less_than(X,Y) :- X<Y.
```

```
cleandatabase :- checkabolish(agenda,4), checkabolish(usedstate,2),
  checkabolish(beststate,1), checkabolish(counter,1).


checkabolish(P,N) :- abolish(P,N), !.
checkabolish(P,N).


measurework :- countup(agenda(X,Y,C,D),NA),
  countup(usedstate(S,C),NB), write(NA),
  write(' incompletely examined state(s) and '),
  write(NB),write(' examined state(s).'), !.


countup(P,N) :- asserta(counter(0)), call(P), counter(K),
  retract(counter(K)), K2 is K+1, asserta(counter(K2)), fail.
countup(P,N) :- counter(N), retract(counter(N)), !.


append([],L,L).
append([I|L1],L2,[I|L3]) :- append(L1,L2,L3).
```

Implementing search with heuristics (*)

Heuristics define a policy for choosing things from the agenda in search. So searching with heuristics can be implemented by redefining the **pick_best_state** in the best-first program, and leaving the rest of the program unchanged. Specifically, it should pick the first state on the agenda for which no other state is preferred by heuristics:

```
pick_best_state(State,Pathlist) :-
  agenda(State,Pathlist,E), not(better_pick(State,Pathlist,E)), !.
better_pick(X,Y,E) :- agenda(X2,Y2,E2), not(X=X2), prefer(X2,Y2,E2,X,Y,E).
```

This same idea was used to implement meta-rules in Section 7.16, and a heuristic *is* a kind of meta-rule, a rule guiding usage of **successor** rules. We'll broaden the concept of a heuristic here to allow use of evaluation functions in its decision, to be fully general.

Such *extended heuristics* can be defined with a **prefer** predicate of six arguments: two groups of three where the first group represents an agenda item preferred to an agenda item represented by the second group. Each group represents the three arguments to agenda facts: state, path list, and evaluation-function value. (The last can be left unbound and ignored if numeric evaluation is hard.) So

```
prefer(X1,Y1,E1,X2,Y2,E2) :- E1 < E2.
```

implements best-first search, and

```
prefer(X1,Y1,E1,X2,Y2,E2) :- length(Y1,L1), length(Y2,L2), L1 < L2.
length([],0).
length([X|L],N) :- length(L,N2), N is N2+1.
```

implements breadth-first search, and

```
prefer(X1,Y1,E1,X2,Y2,E2) :- fail.
```

(or equivalently, no **prefer** rule at all) implements depth-first search. (Then the first agenda item will always be picked; it will always be the successor found most recently, thanks to the **asserta** in **add_state**.) These **prefer** rules can also refer to quite specific characteristics of a search problem, though then they can't be reused in other search problems as much. For instance for the bolts problem, we want part **c** on part **e**, so a heuristic would be to prefer a state in which the top of **e** is clear:

```
prefer(X1,Y1,E1,X2,Y2,E2) :- cleartop(e,X1), not(cleartop(e,X2)).
```

We can have several extended heuristics for a problem, and they will be tried in the order given. For instance we can do a best-first search with a breadth-first criterion to break ties between states with the same evaluation-function value:

```
prefer(X1,Y1,E1,X2,Y2,E2) :- E1 < E2.
prefer(X1,Y1,E,X2,Y2,E) :- length(Y1,L1), length(Y2,L2), L1 < L2.
length([],0).
length([X|L],N) :- length(L,N2), N is N2+1.
```

We could also combine breadth-first and best-first search in a different way:

```
prefer(X1,Y1,E1,X2,Y2,E2) :- Ep1 is E1+1, Ep1 < E2.
prefer(X1,Y1,E1,X2,Y2,E2) :-
  length(Y1,L1), length(Y2,L2), L1 < L2.
```

This says to prefer states according to the evaluation function if they are more than one unit apart, but otherwise pick the state closest to the start.

Sets of heuristics or extended heuristics can exhibit complex behavior; they can be an extensive rule-based system on their own. The most important heuristics (**prefer** rules) should be first, then general-purpose heuristics, and finally default heuristics. As rule-based system rules go, heuristics are powerful because they can be quite general: we can often reuse the same heuristic in many different problems.

The preceding heuristics choose among states. Other heuristics choose among branches of (operators applied to) a state, but these can often be transformed into the first kind by having them reference characteristics of the result of the desired branch.

Compilation of search (*)

Since a search problem can be seen as a rule-based system with **successor** rules, it isn't surprising that we can use compilation techniques for rule-based systems to make search faster. We can exploit all the "compiled forms" in the lower half of Figure 6-1, except virtual facts and decision lattices which require questioning (this is uncommon in search).

Caching is helpful and simple to apply. An agenda is one form of a cache, but we can extend an agenda to include entries from previous related problems. So as we solve problems, we gradually build up a library of solutions, some to full problems and some to subproblems, to make new problems easier. If this creates a large agenda, we can index or hash items for fast lookup. If this is still too slow or requires too much storage, we can cache selectively, using as criterion the amount of work needed to compute the item and perhaps statistics on how often it was used in problems. Caching criteria can also be suggested by the problem itself, as in route planning in a city for which it makes sense to cache routes between landmarks.

Partitioning or modularization is important in many areas of computer science, and search is no exception. Groups of related operators can be put into partitions, each loaded as needed. File loads can be done by a **consult** last in a **successor** rule. Partitions are essential for a system with many **successor** rules, like a general-purpose car repair system, for which quite different parts of the car require quite different repair techniques, so many operator have narrow applicability.

Parallelism can increase the speed of successor rules too. All four types of parallelism discussed in Section 6.9 can be used: partition parallelism, "and" parallelism, "or" parallelism and variable-matching parallelism. Actual concurrency in execution is essential for speed in real-time applications. Good examples are the lattices used for high-performance speech-understanding systems, a kind of and-or-not lattice representing parallelism taken to the maximum extent. For a fixed vocabulary of words, the lattice corresponds to every possible sequence in which sounds can occur to express grammatical utterances using those words. Each node in the lattice represents a sound, and each branch represents a transition to a possible next sound. The speech to be understood is first broken into small varying-length segments representing homogeneous (constant-nature) intervals in the speech. A cost function is used that measures the "distance" any segment comes to some "ideal" sound. Speech recognition then becomes an optimal-path search problem, in which we want to find a path through a lattice that minimizes the sum of the branch costs. Such precompiled search lattices have many other real-time applications.

.SH Keywords:

```
frame problem
agenda of states
the bagof predicate
the cut predicate
iteration
```

Exercises

10-1. (R,A,P) (a) Improve the depth-first search program to print out the answer path list in the forward (opposite) order. Print one state per line.

(b) Suppose we want a search answer to include names of operators used at each step in the solution. Explain how you can do this by writing the problem-dependent files in a certain way without making any changes at all to the problem-independent files.

(c) Suppose states are lists of facts in which the order of the facts doesn't matter--that is, states are described by sets of facts, not lists. Modify the depth-first search program to prevent going to "permutation" states that have the same facts as an already-examined state but in a different order.

10-2. (P) Country X and country Y are carefully negotiating an arms agreement. Each has missiles of varying *strengths, each strength summarized by a single number. Each step in the negotiation does one of two things: (1) eliminates from both sides a single missile of the same numerical strength, or (2) eliminates from one side a missile of strength S and from the other side two missiles whose strength sums to S. Write a program that figures out a sequence of such negotiation steps that will eliminate all the missiles entirely from one or both of the sides, but making sure in the process that no side ever has more than one more missile than the other.*

Use the depth-first program to do searching. Assume to start:

> *Country X: missile strengths 9, 11, 9, 3, 4, 5, 7, 18*
> *Country Y: missile strengths 12, 2, 16, 5, 7, 4, 20*

Type semicolons to your program to get alternative solutions.

Hints: use the ***member*** and **delete** predicates from Sections 5.5 and 5.6, and define a **twomembers** which says two things are different items from the same list.

10-3. Suppose we modify the bolts problem in Section 10.2 a bit. Suppose we have the same operators and starting state, but our goal is to get part **b** and part **c** on bolt 1, with **c** above **b** instead of vice versa.

(a) Give the **goalreached** definition.

(b) Give a corresponding **eval** definition.

(c) What answer will breadth-first search find?

10-4. (P) We didn't show the bolts problem being solved with depth-first search, and there's a good reason: it takes a long time to solve it, a thousand times more perhaps than breadth-first.

(a) Study the program to see why it's taking so much time for such a simple problem. Insert additional code to print out what it's doing periodically.

(b) One possible reason is the lack of unordered sets as opposed to lists in Prolog. Explain why this is a problem. Then explain how you would change the problem-independent code for depth-first search to eliminate this problem. Do so and see if efficiency improves.

(c) Another suggestion is to use heuristics to control the depth-first search. But the implementation of heuristics given in Section 10.11 isn't very helpful. Why not?

(d) Explain how, in place of meta-rules or heuristics, additional successor rules can be written. Do this for the bolts program and analyze the improvement in performance.

10-5. (R,A,P) Load into the Prolog interpreter the breadth-first program given in Section 10.5 and the following additional code:

```
glassesdepth(Cap1,Cap2,Goal,Ans) :- checkretract(cap1(C1)),
  checkretract(cap2(C2)), checkretract(goalvolume(G)),
  checkassert(cap1(Cap1)), checkassert(cap2(Cap2)),
  checkassert(goalvolume(Goal)), !, depthsearch([0,0],Ans).


glassesbreadth(Cap1,Cap2,Goal,Ans) :- checkretract(cap1(C1)),
  checkretract(cap2(C2)), checkretract(goalvolume(G)),
  checkassert(cap1(Cap1)), checkassert(cap2(Cap2)),
  checkassert(goalvolume(Goal)), !, breadthsearch([0,0],Ans).


goalreached([Goalvolume,V2]) :- goalvolume(Goalvolume).
goalreached([V1,Goalvolume]) :- goalvolume(Goalvolume).


successor([V1,V2],[Vsum,0]) :- V2 > 0, Vsum is V1 + V2,
  cap1(Cap1), Vsum =< Cap1.
successor([V1,V2],[0,Vsum]) :- V1 > 0, Vsum is V1 + V2,
  cap2(Cap2), Vsum =< Cap2.
successor([V1,V2],[Cap1,Vdiff]) :- V2 > 0, V1 >= 0,
  cap1(Cap1), Vdiff is V2 - ( Cap1 - V1 ), Vdiff > 0.
successor([V1,V2],[Vdiff,Cap2]) :- V1 > 0, V2 >= 0,
  cap2(Cap2), Vdiff is V1 - ( Cap2 - V2 ), Vdiff > 0.
successor([V1,V2],[Cap1,V2]) :- cap1(Cap1).
successor([V1,V2],[V1,Cap2]) :- cap2(Cap2).
successor([V1,V2],[0,V2]).
successor([V1,V2],[V1,0]).


checkassert(S) :- call(S), !.
checkassert(S) :- asserta(S).


checkretract(S) :- call(S), retract(S), fail.
checkretract(S).
```

This code defines a famous puzzle known as the water-glass problem. You have two empty glasses of capacities C1 and C2, a faucet to fill the glasses with water, and a drain. The glasses are not marked along the sides so you can't fill them a specified partial amount. The goal is to get a certain fixed quantity of liquid. To do this you can fill the glasses, empty the glasses, pour all of one glass into another, or fill one glass from the other. To run the program, type

```
?- glassesbreadth(<capacity-glass-1>,<capacity-glass-2>,<goal-amount>,<answer>).
```

for which the first three arguments are inputs, numbers you must fill in, and the last is the output.

(a) First study the situation in which you have two glasses of sizes 5 and 7. Call the predicate **glassesbreadth** with goal amounts equal to 0, 1, 2, 3, 4, 5, 6, 7, and 8, and construct a diagram of all possible states and their connections. Type the semicolon to obtain all distinct solutions. Note that <**answer**> represents a stack (i.e., it's in reverse chronological order). Describe the overall pattern in the diagram.

(b) Now use the depth-first program given in the chapter. Invoke it with

```
?- glassesdepth
  (<capacity-glass-1>,<capacity-glass-2>,<goal-amount>,<answer>).
```

Try it on all the situations considered previously, and draw its state diagram. Describe how the diagram is different from that for breadth-first search. In particular, how does the depth-first state diagram relate to the breadth-first diagram?

(c) Explain what each of the eight **successor** rules does.

(d) Modify the preceding code defining the operators so that the solutions found will have the desired quantity in the left glass (that is, the glass with the first-mentioned capacity). Test your code with the breadth-first search code.

10-6. (H,P) *Beam search* is a more efficient kind of breadth-first search, a sort of cross between breadth-first search and best-first search. Rather than putting all the successors of a given state on the agenda, it only puts the K best successors according to the evaluation function. Modify the breadth-first search program to do beam search.

10-7. (A,P) Modify the A* search program to do a branch-and-bound search.

10-8. In the city-route planning example of Section 10.1, the last line of the cost function definition seems very dangerous: it would seem to define an infinite loop. Such definitions of commutativity for predicates always have such dangers.

(a) What advantage is gained by having such a line in the definitions?

(b) Why can the danger be ignored for this particular cost function definition used with the A* search program?

(c) Suppose we are doing route planning in which the routes are highways and the nodes are cities. If you can go from city X to city Y along route R you can also go from Y to X backward along R. So state transitions would be commutative, and we could write a single commutativity rule to state this, like the commutativity rule in the cost function definition. Why wouldn't this be a good idea, when it generally is a good idea with costs?

10-9. (R,P) (a) Consider a housekeeping robot that must clean a two-room suite (Figure 10-8). It must vacuum the floors, dust, and empty the trash baskets. Vacuuming the floor generates dust that goes into the trash basket of the room it is in. Emptying the trash baskets in each room requires a trip to the trash chute. Dusting puts dust on the floor, requiring vacuuming.

Assume the following energy costs:

> --10 units to vacuum a single room (assume the robot has a built-in vacuum);
> --6 units to dust a room (assume the robot has a built-in duster);
> --3 units to pick up a trash basket (assume the robot can hold more than one);
> --1 unit to put down a trash basket;
> --3 units to travel between the two offices;
> --8 units to travel between an office and the trash chute (assume there are doors to the trash chute from both rooms);
> --5 units to dispose of the contents of a trash basket down the trash chute.

Use the following reasonable preconditions for actions (and no others):

> --It doesn't make sense to vacuum or dust if the room isn't vacuumable or dusty respectively.
> --It doesn't make sense to dust if a room is vacuumable.
> --It doesn't make sense to pick up a trash basket if a room is vacuumable or dusty.
> --It doesn't make sense to put down something you just picked up.

For the starting state, assume the robot is at the trash chute. Assume that both rooms need dusting. Office 1 has trash in its trash basket, but Office 2 doesn't; and the carpet in Office 1 does not need vacuuming, while the carpet in Office 2 does. The goal is to get everything dusted, every carpet vacuumed, and every basket emptied. The robot must finish at either room but not at the trash chute.

Set this problem up for the A* search program for this chapter. Give appropriate definitions, including a good evaluation function, one that gives a lower bound on the cost to a goal state. You may find you run out of space; if so, revise the program to eliminate unnecessary successors. Run the program and show your results.

Hints: make the first item in each state represent the last operator applied, or "none" for the starting state. Have the cost function use these to assign costs. You don't need **append** or **=** anywhere in your program; use the left sides of rules to get their effect. And in defining **successor**, it may improve efficiency to rule out new states in which the facts are just permutations (ignoring the last-operator-applied item) of the facts in some other state already on the agenda or already visited, as in Exercise 10-1(c).

10-10. (A,P) Combining chemicals to react and make new chemicals is a kind of search. Find a good way to represent states and successor functions for operations including these:

> If we combine one mole of |Cl sub 2| and one mole of |H sub 2 O|, you get one mole of |H Cl O| and one mole of |H Cl|.
>
> If you combine one mole of |Ca C O sub 3| with two moles of |H Cl|, you get one mole of |Ca Cl sub 2|, one mole of |C O sub 2|, and one mole of |H sub 2 O|.
>
> If you combine two moles of |H sub 2 O sub 2| and one mole of |Mn O sub 2|, you get two moles of |H sub 2 O|, one mole of |O sub 2|, and one mole of |Mn O sub 2| (that is, manganese dioxide is only a catalyst).
>
> If you combine one mole of |H sub 2| and one mole of |Cl sub 2|, you get two moles of |H Cl|.
>
> If you combine four moles of |H Cl| with one mole of |Mn O sub 2|, you get one mole of |Mn Cl sub 2|, two moles of |H sub 2 O|, and one mole of |Cl sub 2|.

Hint: represent states by lists of lists.

(b) Write Prolog code, and try it with the depth-first program. Do the problem of making 1 mole of |Ca Cl sub 2| from a starting state consisting of 2 moles of |Cl sub 2|, 1 mole of |Mn O sub 2|, 1 mole of |Ca C O sub 3|, and 2 moles of |H sub 2 O sub 2|. That is, find a sequence of operations that could lead to a mole of |Ca Cl sub 2| being formed--not necessarily the only way it could be formed. Hint: use the **member** predicate of Section 5.5, and the **delete** predicate of Section 5.6 or the **deleteone** of Section 10.7.

(c) Describe a cost function that would be meaningful for this kind of search problem, and describe a corresponding evaluation function.

(d) The Prolog interpreter's conflict-resolution method (trying rules in the order they appear) is not good for chemical-reaction rules. Explain why, and suggest (describe) a better conflict-resolution method from ideas in Chapter 6.

10-11. Suppose you are doing a best-first search in a search space of N states.

(a) Suppose only one state in the search space satisfies the condition for a goal state. What is the smallest maximum size of the agenda during the finding of this goal state?

(b) For the same situation as part (a), what is the largest maximum size of the agenda during the finding of the goal state?

(c) Now suppose there is no goal state in the search space and we must search to confirm this. What is now the smallest maximum size of the agenda in this process of confirmation?

(d) For the situation in part (c), what is the largest maximum size of the agenda?

10-12. (A) Suppose a search space can be represented as a complete (filled-in) and balanced binary tree of 31 states on four levels (not counting the starting state as a level). Suppose there is a single goal state that does lie in the search space.

(a) Suppose we are doing a breadth-first search. What is the maximum possible size of the agenda?

(b) Suppose we are doing a best-first search. What is the maximum possible size of the agenda?

10-13. When a search problem has a solution, breadth-first search, best-first search, and A* search cannot get into an infinite loop. So it's not strictly necessary to check for previously visited states--that just improves the efficiency of search. There are also searches for which the added overhead of checking isn't worth it, and we'd like to define some criteria for recognizing such situations.

(a) Suppose for some search problem there are B successors of every state on the average, fraction P of which are previously visited (either on the agenda or the oldagenda) on the average. How many states can we expect in the search lattice up to and including level K for both visited-state pruning and no pruning at all?

(b) Suppose on the average visited-state checking requires E units of time while all the other processing required of a state (finding it and adding it to the agenda) requires C units on the average. Give an inequality for deciding when visited-state checking pays off.

(c) Now suppose that visited-state checking is not constant time but increases proportionately to the level in the search lattice; let's say the dependency is AK, K the level. What now is the inequality for deciding whether visited-state checking pays off?

10-14.(a) Consider the following program. Suppose it is only used when the first two arguments are bound (inputs) and the third argument is unbound (an output). Could a cut predicate improve efficiency? If so, under what circumstances, and where would you put it? Assume the rules are in a database in the order listed.

```
countup(I,[I|L],N) :- countup(I,L,N2), N is N2+1.
countup(I,[X|L],N) :- countup(I,L,N), not(I=X).
countup(I,[],0).
```

(b) Discuss why the cut predicate is often so tricky and hard to use.

10-15. (A,P) The Micro-Prolog dialect of Prolog has an interesting built-in feature: you can ask it to find and match the Kth rule or fact that is able to match some query. Implement this feature in the "standard" Prolog of this book, using **asserta**. Assume you are given a predicate expression P that is to be queried K times, and assume that P has no side effects so that it is OK to query the previous K-1 matches to it.

10-16. Yet another kind of iteration not yet considered occurs with the "mapcar" feature of Lisp and the vector-processing features of APL. Something similar can be provided for every two-argument function predicate in Prolog. Suppose we have a function predicate **f(X,Y)**. We would want to define a new predicate **f_of_list(L,FL)** such that **FL** is a list created by applying the **f** function to every element of the **L** list in turn. For example, suppose we had a defined predicate **double**:

```
double(X,Y) :- Y is 2 * X.
```

then querying

```
?- double_of_list([1,3,4],L).
```

will result in

```
L=[2,6,8]
```

Write a Prolog definition of **double_of_list**. Explain how the idea can be generalized to other predicates.

10-17. Implement each of the following heuristics in the way described in Section 10.11 using a six-argument **prefer** predicate. Assume they are called on by something like the best-first search program **bestsearch**.

(a) Prefer a state whose fact-list description includes an predicate named **zzxxy** of one argument.

(b) Prefer the state about which the fewest facts are true, assuming states are represented as lists of facts true in them.

(c) Prefer the state whose path list involves the fewest fact additions and deletions along its course. (Assume states are described as lists of facts true in them.)

(d) Prefer any state to one that was a successor to the last state selected.

10-18. Redo Exercise 5-12, but this time have the third argument be the *minimum* distance between two things. Allow for the possibility of cycles in the facts and more than one route between things.

10-19. (E) Suppose we defined a search problem in Prolog, and we now want to try searching backward, as perhaps when the forward search didn't work well. To make things easier, assume there is a single goal state known beforehand.

(a) Give a simple way the definition of the cost function can be modified to provide a cost function for the new backward search.

(b) Give a simple way the evaluation function can be modified to provide an evaluation function for the new problem of searching backward.

(c) Describe in general terms how the successor predicate definitions must be modified. Note that some definitions are easier to use backward than others, so explain which ones these are.

10-20. (P) The program following | REFERENCE 3| plays the game of Reversi. .FS | REFERENCE 3| This and other major programs in this book are available on tape from the publisher. .FE In this game, two players take turns placing pieces on a checkerboard; their pieces are indicated by the symbols **X** and **O**. Players place pieces so as to surround part of a row, column, or diagonal containing an unbroken string of the opponent's pieces; the pieces so surrounded are "captured" and become the opposite (surrounding player's) pieces. A player must always move so as to capture at least one opposing piece; if a player can't move, they forfeit their turn. The game is over when neither player can move; then the player with the most pieces on the board wins. Here's an example of the program output, in which first a human player makes a move, and then the computer.

```
8 X X X X X X X _
7 O O O O X O _ _
6 X O X X X O _ _
5 X O O X X O _ _
4 X O O O X O _ _
4 X O O X O O _ _
2 _ O O X X O _ _
1 _ _ O O X O _ _
  1 2 3 4 5 6 7 8
Give move:76.
```

```
8 X X X X X X X _
7 O O O O X X _ _
6 X O X X X X X _
5 X O O X X X _ _
4 X O O O X O _ _
3 X O O X O O _ _
2 _ O O X X O _ _
1 _ _ O O X O _ _
  1 2 3 4 5 6 7 8
Evaluation of move 12 is 2
Evaluation of move 74 is 193
Evaluation of move 75 is 191
Evaluation of move 86 is 144
Evaluation of move 77 is 178
Evaluation of move 87 is 266
I make move 12
```

In the program, the board is stored as an 8-item list of 8-item lists, where a 0 element means a blank board position, a 1 element means a board position occupied by the human player, and a -1 element means a board position occupied by the computer player. States are represented as 10-item lists: the player who last moved, the move they took, and the eight rows of the resulting board position.

(a) The program shown following uses a simple evaluation function. Redefine **pick_best_move** so it uses nonnumeric heuristics to select a move. That is, don't use any numbers for move selection except to look up board positions by row and column number.

(b) Now change **gamesearch** so the computer can play against itself, using your heuristics for player 1 and its evaluation function for player -1. Make sure by your heuristics that player 1 wins--improve the heuristics as necessary. Then use the heuristics for player -1, and the evaluation function for player 1, and further improve the heuristics as necessary so that player -1 wins.

(c) Now change the **pick_best_move** rules for player -1 so it looks ahead ahead one move to anticipate what the other player will do, and chooses the move leading to the best result assuming player 1 always makes the best choice available to them. Run this smarter player against the heuristics from part (b) used for player 1. Who wins?

(d) Probably player -1 won in part (c), but the comparison wasn't fair because player 1 doesn't look ahead. But they don't need to if they just have better heuristics. Demonstrate this.

```
/* Top-level routines */
go :-
  write('This program plays the game of Reversi (X is you, O is me).'),
  nl, start_state(SS), gamesearch(SS).


start_state([-1,none,Z,Z,Z,[0,0,0,1,-1,0,0,0],[0,0,0,-1,1,0,0,0],Z,Z,Z])
  :- zero_row(Z).


zero_row([0,0,0,0,0,0,0,0]).


gamesearch(State) :- goalreached(State), print_win(State), !.
gamesearch([-1,M|B]) :- not(get_move(B,1,S,NB)),
  write('You cannot move so I get another turn.'), nl, !,
  gamesearch([1,M|B]).
gamesearch([-1,OM|B]) :- print_board(B), write('Give move:'),
  repeatread(Movecode), move_decode(Movecode,Move),
  get_move(B,1,Move,NB), !, gamesearch([1,Move|NB]).
gamesearch([1,M|B]) :-  print_board(B),
  pick_best_move([1,M|B],Newstate),
  write_move(Newstate), !, gamesearch(Newstate).
gamesearch([1,M|B]) :- not(get_move(B,-1,S,NB)),
  not(get_move(B,1,S,NB)), print_win([1,M|B]), !.
gamesearch([1,M|B]) :- not(get_move(B,-1,S,NB)),
  write('I cannot move so you get another turn.'),
  nl, !, gamesearch([-1,M|B]).
gamesearch(S) :- print_win(S), !.

/* Intermediate routines */
pick_best_move(S,NS) :- asserta(ebest(none,1000000)), successor(S,S2),
  eval(S2,E2), write_eval(S2,E2), ebest(S1,E1), E2<E1,
  retract(ebest(S1,E1)), asserta(ebest(S2,E2)), fail.
pick_best_move(S,NS) :- ebest(NS,E), retract(ebest(NS,E)).


goalreached([X,Y|B]) :- not(somezero(B)).


somezero(B) :- member(R,B), member(0,R).


successor([OP,Lastmove|Oldboard],[P,M|Newboard]) :-
  opposite(OP,P), get_move(Oldboard,P,M,Newboard).


opposite(1,-1).
opposite(-1,1).


repeatread(M) :- read(M).
repeatread(M) :- write('Invalid move.  Try again:'),
  nl, repeatread(M).


move_decode(M,[X,Y]) :- X is M//10, Y is M mod 10.

/* Printing routines */
write_move([P,[X,Y]|B]) :- Move is (10*X)+Y, write('I make move '),
  write(Move), nl, !.
write_eval([P,[X,Y]|B],E) :- write('Evaluation of move '),
  Movecode is (10*X)+Y, write(Movecode), write(' is '),
  write(E), nl, !.


print_win([P,M|B]) :- print_board(B),
  countup(1,B,N1), countup(-1,B,N2), print_win2(N1,N2).
print_win2(N1,N2) :- N2<N1, write('Congratulations! You win.'), nl.
print_win2(N1,N2) :- N2>N1, write('Sorry--I win.'), nl.
print_win2(N,N) :- write('The game is a draw.'), nl.


countup(I,[],0) :- !.
countup(I,[R|B],N) :- countrow(I,R,N2), countup(I,B,N3), N is N2+N3.
```

```
countrow(I,[],0) :- !.
countrow(I,[I|L],N) :- countrow(I,L,N2), N is N2+1, !.
countrow(I,[J|L],N) :- countrow(I,L,N).


print_board(B) :- reverse(B,RB), print_board2(RB,8), !.
print_board2([],N) :- write('  1 2 3 4 5 6 7 8'), nl.
print_board2([Row|Board],N) :- write(N), print_row(Row), N2 is N-1,
  print_board2(Board,N2).
print_row([]) :- nl.
print_row([I|Row]) :- decode(I,DI), write(' '), write(DI),
  print_row(Row).


decode(0,'_').
decode(1,'X').
decode(-1,'O').

/* Move generation */
get_move(B,P,[X,Y],NB) :- get_move2(B,[X,Y],FR,BR,FI,BI),
  fix_key_row_half(P,FI,NFI), fix_key_row_half(P,BI,NBI),
  fix_rows(FR,X,P,NFR), fix_rows(BR,X,P,NBR),
  not(nochange(FI,NFI,BI,NBI,FR,NFR,BR,NBR)),
  reverse(NFI,RFI), reverse(NFR,RFR),
  append(RFI,[P|NBI],Row), append(RFR,[Row|NBR],NB).


nochange(X,X,Y,Y,A,B,C,D) :- samefront(A,B), samefront(C,D).


samefront([],[]) :- !.
samefront([X|L],[X|M]).


get_move2(B,M,FR,BR,FI,BI) :- get_move3(B,1,[],M,FR,BR,FI,BI).


get_move3([R|B],Y,FR,[X,Y],FR,B,FI,BI) :- get_move4(R,1,[],X,FI,BI).
get_move3([R|B],Y,FR,M,FRF,BR,FI,BI) :-  Y2 is Y+1,
  get_move3(B,Y2,[R|FR],M,FRF,BR,FI,BI).


get_move4([0|R],X,FI,X,FI,R).
get_move4([I|R],X,FI,XF,FIF,BI) :- X2 is X+1,
  get_move4(R,X2,[I|FI],XF,FIF,BI).

/* Board update for a particular move */
fix_key_row_half(P,R,NR) :- fix_key_row_half2(P,R,NR), !.
fix_key_row_half(P,R,R) :- !.


fix_key_row_half2(P,[],[]) :- !, fail.
fix_key_row_half2(P,[0|L],[0|L]) :- !, fail.
fix_key_row_half2(P,[P|L],[P|L]) :- !.
fix_key_row_half2(P,[OP|L],[P|L2]) :- opposite(P,OP),
  fix_key_row_half2(P,L,L2).


fix_rows([],X,P,[]) :- !.
fix_rows([R],X,P,[R]) :- !.
fix_rows(B,X,P,NB) :- Xm1 is X-1, Xp1 is X+1, opposite(P,OP),
  fix_rows2(B,OP,NB,Xm1,X,Xp1,VL,VM,VR), !.


fix_rows2(B,OP,B,dead,dead,dead,VL,VM,VR).
fix_rows2([],OP,[],XL,XM,XR,VL,VM,VR) :- topfix(OP,VL),
  topfix(OP,VM), topfix(OP,VR).
fix_rows2([R|B],OP,[NR|NB],XL,XM,XR,VL,VM,VR) :-
  fix_rowL(R,OP,1,NR,XL,XM,XR,VL,VM,VR,NXL,NXM,NXR),
  fix_rows2(B,OP,NB,NXL,NXM,NXR,VL,VM,VR).


topfix(OP,OP) :- !.
topfix(OP,V).


fix_rowL(R,OP,X,NR,dead,dead,XR,VL,VM,VR,dead,dead,NXR) :-
  fix_rowR(R,OP,X,NR,XR,VR,NXR).
fix_rowL(R,OP,X,NR,dead,XM,XR,VL,VM,VR,dead,NXM,NXR) :-
  fix_rowM(R,OP,X,NR,XM,XR,VM,VR,NXM,NXR).
fix_rowL(R,OP,X,NR,0,XM,XR,OP,VM,VR,dead,NXM,NXR) :-
  fix_rowM(R,OP,X,NR,XM,XR,VM,VR,NXM,NXR).
fix_rowL([0|R],OP,X,[0|NR],X,XM,XR,OP,VM,VR,dead,NXM,NXR)
  :- Xp1 is X+1, fix_rowM(R,OP,Xp1,NR,XM,XR,VM,VR,NXM,NXR).
fix_rowL([OP|R],OP,X,[VL|NR],X,XM,XR,VL,VM,VR,Xm1,NXM,NXR) :-
  Xp1 is X+1, Xm1 is X-1,
  fix_rowM(R,OP,Xp1,NR,XM,XR,VM,VR,NXM,NXR).
fix_rowL([P|R],OP,X,[P|NR],X,XM,XR,P,VM,VR,dead,NXM,NXR) :-
  Xp1 is X+1, opposite(OP,P),
  fix_rowM(R,OP,Xp1,NR,XM,XR,VM,VR,NXM,NXR).
fix_rowL([I|R],OP,X,[I|NR],XL,XM,XR,VL,VM,VR,NXL,NXM,NXR) :-
  Xp1 is X+1,
  fix_rowL(R,OP,Xp1,NR,XL,XM,XR,VL,VM,VR,NXL,NXM,NXR).


fix_rowM(R,OP,X,NR,dead,XR,VM,VR,dead,NXR) :-
  fix_rowR(R,OP,X,NR,XR,VR,NXR).
```

```
fix_rowM([0|R],OP,X,[0|NR],X,XR,OP,VR,dead,NXR) :-  Xp1 is X+1,
  fix_rowR(R,OP,Xp1,NR,XR,VR,NXR).
fix_rowM([OP|R],OP,X,[VM|NR],X,XR,VM,VR,X,NXR) :-  Xp1 is X+1,
  fix_rowR(R,OP,Xp1,NR,XR,VR,NXR).
fix_rowM([P|R],OP,X,[P|NR],X,XR,P,VR,dead,NXR) :-  Xp1 is X+1,
  opposite(OP,P), fix_rowR(R,OP,Xp1,NR,XR,VR,NXR).
fix_rowM([I|R],OP,X,[I|NR],XM,XR,VM,VR,NVM,NXR) :-  Xp1 is X+1,
  fix_rowM(R,OP,Xp1,NR,XM,XR,VM,VR,NVM,NXR).


fix_rowR(R,OP,X,R,dead,VR,dead).
fix_rowR(R,OP,X,R,9,OP,dead).
fix_rowR([0|R],OP,X,[0|R],X,OP,dead).
fix_rowR([OP|R],OP,X,[VR|R],X,VR,Xp1) :- Xp1 is X+1.
fix_rowR([P|R],OP,X,[P|R],X,P,dead) :- opposite(OP,P).
fix_rowR([I|R],OP,X,[I|NR],XR,VR,NXR) :- Xp1 is X+1,
  fix_rowR(R,OP,Xp1,NR,XR,VR,NXR).

/* Evaluation function */
eval([P,[X,Y],A,B,C,D,E,F,G,H],Ev) :- eval2s(A,B,Ev1),
  eval2s(H,G,Ev2), eval2m(C,Ev3), eval2m(D,Ev4),
  eval2m(E,Ev5), eval2m(F,Ev6),
  Ev is -P*(Ev1+Ev2+Ev3+Ev4+Ev5+Ev6), !.


eval2m([A,B,C,D,E,F,G,H],Ev) :-
  Ev2 is ((30*A)+(-15*B)+C+D+E+F+(-15*G)+(30*H)),
  sc(A,B,C1), sc(H,G,C2), Ev is Ev2+C1+C2, !.


sc(0,B,0).
sc(A,0,0).
sc(1,1,30).
sc(1,-1,20).
sc(-1,-1,-30).
sc(-1,1,-20).


eval2s([A,B,C,D,E,F,G,H],[I,J,K,L,M,N,O,P],Ev) :-
  Ev2 is
   (200*A)+(-50*B)+(30*C)+(30*D)+(30*E)+(30*F)+(-50*G)+(200*H),
  Ev3 is
   -((50*I)+(100*J)+(15*K)+(15*L)+(15*M)+(15*N)+(100*O)+(50*P)),
  sc2(A,I,J,B,Ev4), sc2(H,G,O,P,Ev5), sc(C,K,Ev6), sc(D,L,Ev7),
  sc(E,M,Ev8), sc(F,N,Ev9),
  Ev is Ev2+Ev3+Ev4+Ev5+Ev6+Ev7+Ev8+Ev9, !.


sc2(0,B,C,D,0).
```

```
sc2(A,B,C,D,Ev) :- sc(A,B,E1), sc(A,C,E2), sc(A,D,E3),
  Ev is 3*(E1+E2+E2+E3).

/* Utility functions */
reverse(L,RL) :- reverse2(L,[],RL), !.


reverse2([],L,L).
reverse2([X|L],L2,RL) :- reverse2(L,[X|L2],RL).


append([],L,L) :- !.
append([X|L],L2,[X|L3]) :- append(L,L2,L3).


member(X,[X|L]).
member(X,[Y|L]) :- member(X,L).
```

Go to book index

Abstraction in search

When people solve without computers the search problems discussed in the last two chapters, they only use the techniques so far described as a last resort. Instead, they try to reason about abstractions, what's known as *hierarchical reasoning*.

Abstraction is simplifying your model of the world so you can reason more easily about it. Intermediate predicates for rule-based systems (see Section 7.5) are one form of abstraction, since they summarize a category of data and provide a handle on that summarization. Abstraction is essential in organizing and managing large numbers of facts, as we will see in the next chapter. Abstraction is also a way to simplify large search problems. Search abstraction works best when a problem is decomposable, and exploits the preconditions and postconditions of operators.

Means-ends analysis

The classic technique for solving search problems by abstraction is means-ends analysis. It applies to those decomposable search problems (see Section 9.12) for which clear "major operators" or "recommended operators" on the solution path can be identified in advance. To specify the these operators for specific situations, means-ends analysis needs something like a table. Usually these tables refer to the difference between the current state and the goal state, and are thus called *difference tables*.

Difference tables describe operators in terms of their preconditions and postconditions (see Section 9.4). So they're akin to the **successor** definitions of Chapter 10, but with one big difference: difference tables only *recommend* an operator appropriate to a state and a goal, with no concern if the operator can actually be applied to the state. Difference tables provide a way of decomposing a problem into three simpler subproblems: a subproblem of getting from the current state to a state in which we can apply the recommended operator, a subproblem of applying the operator, and a subproblem of going from there to the goal state. Formally, a search from state S to some goal state G is decomposed into:

1. satisfying the preconditions (prerequisites) for some recommended operator O when starting from S, by going to state S2;
2. applying operator O to S2 to get to state S3 (by postconditions);
3. going from S3 to the goal state G.

The first and third steps are search problems themselves, possibly requiring additional decompositions of their own by the difference table. So means-ends analysis is recursive search. Means-ends analysis is also hierarchical reasoning because we start with big wholes and gradually reason down to little details. The ends truly justify the means.

Complete and correct specification in the difference table of preconditions and postconditions of operators is essential for means-ends analysis. In particular, you must carefully distinguish preconditions from the looser conditions recommending operators. Both could be mixed together in the **successor** rules of Chapter 10, but that won't work here.

A simple example

Means-ends analysis is useful for many human "planning" activities. Here's a simple example of planning for an office worker. Suppose we have a difference table of three rules (see Figure 11-1), written informally as:

> If in your current state you are hungry, and in your goal state you are not hungry, then either the "visit_cafeteria" or "visit_vending_machine" operator is recommended.
>
> If in your current state you do not have change, and if in your goal state you have change, then the "visit_your_office" operator or the "visit_secretary" operator is recommended.
>
> If in your current state you do not know where something is, and in your goal state you do know, then either the "visit_your_office", "visit_secretary", or "visit_colleague" operator is recommended.

We need preconditions and postconditions for the preceding operators:

> Preconditions of visit_cafeteria: you know where the cafeteria is.
> Postconditions of visit_cafeteria: you are no longer hungry.
> Preconditions of visit_vending_machine: you know where the vending machine is and you have change.
> Postconditions of visit_vending_machine: you are no longer hungry.
> Preconditions of visit_your_office: none (we assume everyone knows where his or her office is).
> Postconditions of visit_your_office: you have change and you know where everything is (since you can look it up in, say, the phone directory).
> Preconditions of visit_secretary: you know where the secretary is and the secretary is available.
> Postconditions of visit_secretary: you have change, and you know where everything is.
> Preconditions of visit_colleague: none.
> Postconditions of visit_colleague: you know where everything is.

Since we will implement difference tables in Prolog, we'll follow the usual top-to-bottom and left-to-right conventions about order of action. So we will try difference-table rules in the order listed, try operator alternatives in the order listed, and try to satisfy preconditions and postconditions in the order listed. So the order of rows and columns in Figure 11-1 is significant.

Let's illustrate means-ends analysis for the situation in which you are hungry, you have no change, a secretary is available, and you are new and don't know where anything is besides your own office; and your goal is any state in which you are not hungry. (Other things can be true of your final state besides being not hungry, but hunger relief is what you insist on.)

> 1. The first rule succeeds, so you want to visit the cafeteria.
>
> 2. This has the precondition that you know where the cafeteria is, and since you don't, you must recursively call on means-ends analysis to find a way to change that discrepancy (difference).
>
> 3. Only the third rule can handle this kind of a difference between a state and a goal, so we try to apply the first operator recommended, visit_your_office. (That is, you will look through the telephone directory to locate the cafeteria.)
>
> 4. There are no preconditions for visit_your_office (everybody knows where their office is), so it can be applied without further recursion. And the state after visit_your_office satisfies the preconditions of visit_cafeteria as you hoped--no postconditions will interfere.
>
> 5. The postcondition of visit_cafeteria is that you are no longer hungry, and the goal of the problem was to make this become true. So we're done. So the operator sequence found to solve the problem is visit_your_office and then visit_cafeteria.

To show the importance of difference-table order, suppose we switch the order of visit_cafeteria and visit_vending_machine in the first rule, switch the order of visit_your_office and visit_secretary in the second rule, put visit_colleague first in the third rule, and switch the second and third rules, so we have (see Figure 11-2):

> If in your current state you are hungry and in your goal state you are not hungry, then either the "visit_vending_machine" or "visit_cafeteria" operator is recommended.
>
> If in your current state you do not know where something is, and in your goal state you do know, then either the "visit_colleague", "visit_your_office", or "visit_secretary" operator is recommended.
>
> If in your current state you do not have change and if in your goal state you have change, then the "visit_secretary" operator or the "visit_your_office" operator is recommended.

Let's redo the problem in which you are hungry, you need change, a secretary is available, and you don't know anything, and you want to get to a state in which you are no longer hungry. Means-ends analysis finds a different, but equally valid, solution:

1. The first rule succeeds, so it is recommended that you first try to visit a vending machine. This has two preconditions: that you know where one is and you have change.

2. We recursively call on means-ends analysis to satisfy these preconditions. The starting state is the original starting state, and the new goal is any state in which you know where a vending machine is and you have change.

3. The first rule won't help this difference, but the second rule will, the one that recommends visiting a colleague (to ask where a vending machine is). This has no preconditions, so you can directly apply it.

4. After applying the postconditions of visit_colleague, we must still get to a state in which we have change; remember, visit_vending_machine had two preconditions. (All my colleagues are poor and never have any change--or so they tell me.) So we need a recursive "postcondition search" to accomplish this, from the state in which we are hungry, we need change, a secretary is available, and we know everything. For this difference the third rule is recommended, and the recommended operator is visit_secretary.

5. This has two preconditions: that you know where the secretary is and that secretary is available. Both conditions hold in the current state (the first condition since the beginning, and the second condition was just accomplished by visit_colleague). So we can directly apply visit_secretary.

6. Now taking into account the postconditions of visit_secretary, we're in a state in which we are hungry, we have change, a secretary is available, and we know where everything is. This satisfies all the conditions of the visit_vending_machine operator, so we apply it. Examining its postconditions, we see we've reached a goal state.

7. So the overall operator sequence is to visit your colleague to find out where a vending machine is and where the secretary is, visit the secretary to get change, and then visit a vending machine.

This example didn't require backtracking, but you may need it when your first choice for a rule or operator doesn't pan out, just as with depth-first search. A choice doesn't pan out when no way can be found to satisfy the preconditions or to handle the postconditions. However, the more "intelligent" nature of means-ends analysis compared to the other search methods means that it backtracks a lot less than they do on nicely decomposable problems.

This example of means-ends analysis is highly simplified. If we wanted to realistically model the way people solve daily problems like the preceding, we would need a lot more operators, preconditions, and postconditions. Nevertheless, many psychologists argue that something like means-ends analysis underlies much human activity.

Partial state description

In the preceding example, we described goal states by giving a few facts that must be true in them. For instance, we said the overall goal was a state in which you are no longer hungry. This *partial state description* is very common with means-ends analysis, because it's hard to define a single **goalreached** predicate when each precondition decomposition has a different goal.

Since means-ends analysis is driven by differences between states, we must be careful to define this difference for partial state descriptions. Basically, we can go through the list of facts describing the goal G and "cross out" facts also true for current state S. Here's the definition:

```
difference([],S,[]).
difference([P|G],S,G2) :- singlemember(P,S), !, difference(G,S,G2).
difference([P|G],S,[P|G2]) :- difference(G,S,G2).
```

(Predicate **singlemember** was defined in Section 10.7 and checks only once whether an item is a member of a list; the **!** (cut symbol) was explained in Section 10.7 too.) Then we write difference-table rules to refer to these differences.

So for instance:

```
?-difference
  ([on(b,c),on(e,f),on(a,e)],[on(a,b),on(b,c),on(d,e),on(e,f)],D).
D=[on(a,e)]
```

Implementation of means-ends analysis

With these details out of the way, implementation of means-ends analysis is straightforward. We represent the difference table with rules and/or facts defining

```
recommended(<difference>,<operator>)
```

where **<difference>** is some result (binding of the third argument) of the **difference** predicate defined in the last section, and **<operator>** is the name of a recommended operator. Generally, each operator will have at least one **recommended** definition.

Preconditions are best expressed as two-argument facts. The first argument can be the operator, and the second argument the list of associated preconditions, predicate expressions that must be present in a state description for the operator to be used in that state:

```
precondition(<operator>,[<predexpr1>,<predexpr2>,...]).
```

But postconditions are tricky. An operator can not only make certain conditions true, but can make certain other true conditions become false--otherwise, we couldn't undo anything. So we'll distinguish two kinds of postconditions: predicate expressions added by an operator, and predicate expressions deleted.

```
addpostcondition(<operator>,[<predexpr1>,<predexpr2>...]).
deletepostcondition(<operator>,[<predexpr1>,<predexpr2>...]).
```

By the way, operators can have arguments just as in Chapter 9, and these arguments can be referenced in the precondition and postcondition lists.

That covers the format of the problem-dependent part of means-ends analysis; now for the problem-independent part. We'll use *double recursion*, which sounds like a horrible disease but just means a self-referencing program that self-references twice. Our top-level rule will be a recursive **means_ends** predicate of four arguments:

> **State**, a complete list of facts true in the starting state (an input);
> **Goal**, a list of those facts that must be true in the goal state, a partial description of the goal state (an input);
> **Oplist**, the of operators that will get you from the start to the goal (an output);
> **Goalstate**, the complete list of facts true in the goal state (an output).

Here's the program:

```
/* Problem-independent means-ends analysis */
means_ends(State,Goal,[],State) :- difference(Goal,State,[]), !.
means_ends(State,Goal,Oplist,Goalstate) :-
  difference(Goal,State,D),
  recommended(Dsub,Operator), subset(Dsub,D),
  precondition(Operator,Prelist),
  means_ends(State,Prelist,Preoplist,Prestate),
  deletepostcondition(Operator,Deletepostlist),
  deleteitems(Deletepostlist,Prestate,Prestate2),
  addpostcondition(Operator,Addpostlist),
  union(Addpostlist,Prestate2,Postlist),
  means_ends(Postlist,Goal,Postoplist,Goalstate),
  append(Preoplist,[Operator|Postoplist],Oplist).
```

This recursive program has a single basis step (the first line), which says we don't need any operators to solve a problem in which the starting state includes all the goal facts. The rest of the lines are a single induction step, which you'll note has two recursive calls: the first for the preconditions, the second for the postconditions. The lines say to first compute the list of facts different between **State** and **Goal**, and find a recommended operator for some subset of those facts. Look up the preconditions of the operator and recursively call **means_ends** to figure how to satisfy them. Then look up the deletion postconditions and delete them from the final state resulting from the precondition recursion; look up the addition postconditions, and add them to that state. Now we have the state after application of the original recommended operator. So recursively call **means_ends** to figure how to get from here to the original goal. The final operator list for the whole problem is the appending together of the precondition-recursion operator list, the recommended operator, and the postcondition-recursion operator list.

The **means_ends** program requires five utility functions: **singlemember**, **subset**, **append**, **union**, and **deleteitems**. The **singlemember** (defined in Section 10.7) checks whether an item is a member of a list, while preventing backtracking. The **subset** (defined in Section 5.7) says whether every item in its list first argument is also a member of its list second argument. The **append** (defined in Section 5.6) glues two lists together to create a new list. The **union** is like the **append** except it removes duplicates from the result. The **deleteitems** removes all members of one list from another list, and uses the **delete** predicate of Section 5.6. To improve efficiency, cut symbols **!** appear in some of these definitions.

```
/* Utility functions for means-ends analysis */
singlemember(X,[X|L]) :- !.
singlemember(X,[Y|L]) :- singlemember(X,L).


subset([],L).
subset([X|L],L2) :- singlemember(X,L2), subset(L,L2).


append([],L,L).
append([X|L],L2,[X|L3]) :- append(L,L2,L3).


union([],L,L).
union([X|L1],L2,L3) :- singlemember(X,L2), !, union(L1,L2,L3).
union([X|L1],L2,[X|L3]) :- union(L1,L2,L3).


deleteitems([],L,L).
deleteitems([X|L],L2,L3) :- delete(X,L2,L4), deleteitems(L,L4,L3).


delete(X,[],[]).
delete(X,[X|L],M) :- !, delete(X,L,M).
delete(X,[Y|L],[Y|M]) :- delete(X,L,M).
```

To make the means-ends program more user-friendly, it's helpful to put in some error checking. This goes after the two original definitions of **means_ends**.

```
means_ends(State,Goal,Oplist,Goalstate) :-
  not(difference(Goal,State,D)),
  write('Bug found: no difference defined between '),
  write(Goal), write(' and '), write(State), nl, !, fail.
means_ends(State,Goal,Oplist,Goalstate) :-
  difference(Goal,State,D), recommended(Dsub,Operator),
  subset(Dsub,D), not(precondition(Operator,Prelist)),
  write('Bug found: no preconditions given for operator '),
  write(Operator), nl, !, fail.
means_ends(State,Goal,Oplist,Goalstate) :-
  difference(Goal,State,D),
   recommended(Dsub,Operator), subset(Dsub,D),
  not(deletepostcondition(Operator,Deletepostlist)),
  write('Bug found: no deletepostconditions given for operator '),
  write(Operator), nl, !, fail.
means_ends(State,Goal,Oplist,Goalstate) :-
  difference(Goal,State,D),
  recommended(Dsub,Operator), subset(Dsub,D),
  not(addpostcondition(Operator,Addpostlist)),
  write('Bug found: no addpostconditions given for operator '),
  write(Operator), nl, !, fail.
```

A harder example: flashlight repair

Car repair is often a decomposable search problem and thus appropriate for means-ends analysis, but it's too complicated to use as an example here. So let's consider repair of a flashlight, the kind shown in Figure 11-3. This flashlight has two batteries inside a case, attached to a light bulb. To reach the batteries you must disassemble the case, and to reach the light bulb you must disassemble the top as well as disassembling the case. We'll use the following facts to describe states in flashlight repair (which mean what they suggest):

```
defective(batteries).
ok(batteries).
defective(light).
ok(light).
open(case).
closed(case).
unbroken(case).
broken(case).
```

```
open(top).
closed(top).
inside(batteries).
outside(batteries).
```

We'll assume the following operators: replace_batteries, replace_light, disassemble_case, assemble_case, disassemble_top, assemble_top, turn_over_case, and smash_case.

Now let's apply our means-ends program to the flashlight-repair problem. First we need a difference table (summarized in Figure 11-4), stating recommended operators with facts of the predicate name **recommended**:

```
recommended([ok(batteries)],replace_batteries).
recommended([ok(light)],replace_light).
recommended([open(case)],disassemble_case).
recommended([open(case)],smash_case).
recommended([open(top)],disassemble_top).
recommended([open(top)],smash_case).
recommended([closed(case)],assemble_case).
recommended([closed(top)],assemble_top).
recommended([outside(batteries)],turn_over_case).
recommended([outside(batteries)],smash_case).
```

As you may recall, the first argument to **recommended** is part of the difference of the goal from the current state, and the second argument is the recommended operator. So the first argument is what we want to become true, what isn't true in the starting state. So read the first fact as "If you want to get the batteries to be OK because they aren't now, try the replace_batteries operator." All these recommendations refer to single-fact desires, but that's only because it's hard to do two things at once with one action on a flashlight; other kinds of repair are different.

The order of the **recommended** rules is important, because they'll be considered by the Prolog interpreter in that order, and the first one applying to the given state and goal will be the first one used. So we've put in front those for the most important operators, the "replace_batteries" and "replace_light" operators. Several situations suggest the "smash_case" operator besides another operator, but we want to make "smash_case" a last resort, and hence put the line mentioning it after the line for the alternative. Now for the preconditions:

```
precondition(replace_batteries,
  [open(case),outside(batteries),unbroken(case)]).
precondition(replace_light,[open(top)]).
precondition(disassemble_case,[closed(case)]).
precondition(assemble_case,
  [open(case),closed(top),unbroken(case)]).
```

```
precondition(disassemble_top,[open(case),closed(top)]).
precondition(assemble_top,[open(top)]).
precondition(turn_over_case,[open(case)]).
precondition(smash_case,[]).
```

Notice we don't need to list everything that must be true for an operator to be applied, only the major conditions that imply everything else. For instance, the precondition for "replace_light" that the top be open requires that the case be open first, but we will find this in looking up the preconditions to the disassemble_top operator anyway. So means-ends analysis lets us put preconditions in the places that make our programs clearest.

Now for the postconditions. Often the facts deleted by postconditions (deletepostconditions) are just the preconditions, and often the facts added by postconditions (addpostconditions) are just the opposites of all the deletepostconditions. So postconditions are often easy to write. But this isn't true when operators have "side effects", like the "smash_case" operator that does several things simultaneously--and that's one of the reasons it's not a very good operator.

```
deletepostcondition(replace_batteries,
  [outside(batteries),defective(batteries)]).
deletepostcondition(replace_light,[defective(light)]).
deletepostcondition(disassemble_case,[closed(case)]).
deletepostcondition(assemble_case,[open(case)]).
deletepostcondition(disassemble_top,[closed(top)]).
deletepostcondition(assemble_top,[open(top)]).
deletepostcondition(turn_over_case,[inside(batteries)]).
deletepostcondition(smash_case,
  [unbroken(case),closed(case),closed(top),inside(batteries)]).


addpostcondition(replace_batteries,
  [inside(batteries),ok(batteries)]).
addpostcondition(replace_light,[ok(light)]).
addpostcondition(disassemble_case,[open(case)]).
addpostcondition(assemble_case,[closed(case)]).
addpostcondition(disassemble_top,[open(top)]).
addpostcondition(assemble_top,[closed(top)]).
addpostcondition(turn_over_case,[outside(batteries)]).
addpostcondition(smash_case,
  [broken(case),open(case),open(top),outside(batteries)]).
```

Note that even if some **deletepostcondition** facts are not true for a state, no harm is done--those facts will just be ignored--because the **delete** predicate and hence the **deleteitems** predicate always succeed, even when they don't delete anything.

Running the flashlight program

Let's run our flashlight and means-ends analysis code for the situation in which the batteries are defective and we want to get the flashlight working again. The correct solution is shown in Figure 11-5. Let's assume the flashlight is assembled and unbroken when we start out, so the starting state is

```
[closed(case),closed(top),inside(batteries),defective(batteries),
  ok(light),unbroken(case)].
```

The goal is any state in which these facts are true (i.e., this partial goal description holds):

```
[ok(batteries),closed(case),closed(top)].
```

(A flashlight isn't much good unless it's put back together again, hence the last two conditions.) To do means-ends analysis, we supply the preceding two lists as inputs (the first two arguments) to **means_ends**, leaving the last two arguments unbound:

```
?- means_ends([closed(case),closed(top),inside(batteries),
  defective(batteries),ok(light),unbroken(case)],
  [ok(batteries),closed(case),closed(top)], Operators, Final_state).
```

What will happen is summarized in Figures 11-6 (an overview) and 11-7 (the details). The four arguments to **means_ends** (**State, Goal, Oplist, and Goalstate**) are listed for each call, plus the local variable **Operator**. The calls are drawn like the predicate hierarchies of Figures 10-5, 10-6, and 10-7. **Oplist** and **Goalstate** are the outputs bound as a result of reasoning, so the values shown for them are their final bindings.

In more detail, here's what Figure 11-7 recounts. Fasten your seat belts.

1. The **difference** function is applied to the original goal and the starting state. The last two facts in the goal cancel out, leaving only **[ok(batteries)]**. For this difference, "replace_batteries" is the recommended operator. The problem can then be recursively divided into three parts: satisfying the preconditions of replace_batteries, applying it, and going from there to the original goal.

2. The three preconditions of replace_batteries are that case is open, the batteries are outside the case, and the case is unbroken. So we recursively call the predicate **means_ends** with first argument the original starting state, and second argument (partial goal description) the precondition list

```
[open(case),outside(batteries),unbroken(case)].
```

3. The difference between this new goal and the original starting state is

```
[open(case),outside(batteries)].
```

For this, either the "disassemble_case", "smash_case", or "turn_over_case" operators are recommended. We always try the first recommendation first. That's disassemble_case, and its only precondition is that the case be closed.

4. So we do a third level of recursive call with the same starting state and the goal

```
[closed(case)].
```

But this is already true in the starting state. So the difference is **[]**, and no work must be done to solve this recursion. We succeed with the first **means_ends** rule (the basis step), setting the third argument (the operator list) to **[]** and the fourth argument (the final state) to the starting state. Such trivial search problems are indicated in Figure 11-7 by the words "nothing needed".

5. Now we return to the second level of recursion where we are trying to apply the recommended "disassemble_case" operator. We just discovered that no work is needed to satisfy preconditions, so we just figure out the consequences of disassemble_case. The facts to be deleted (the deletepostconditions) are **[closed(case)]**, and the facts to be added (the addpostconditions) are **[open(case)]**. So the state after applying disassemble_case (the **Postlist** variable) is

```
[open(case),closed(top),inside(batteries),defective(batteries),
  ok(light),unbroken(case)]
```

6. Now we recurse (to the third level) to find how, from this new state, we can satisfy the second-level partial goal description

```
[open(case),outside(batteries),unbroken(case)].
```

The difference is

```
[outside(batteries)].
```

so now only one fact is now different. For this difference, two operators are recommended: turn_over_case and smash_case, in that order. We try the first one first.

7. Turn_over_case has precondition

```
[open(case)].
```

which is already true. So the precondition recursion for turn_over_case succeeds trivially. Using the deletepostconditions and addpostconditions, the new state after turn_over_case is

```
[outside(batteries),open(case),closed(top),
  defective(batteries),ok(light),unbroken(case)]
```

We now compare this to the second-level goal of

```
[open(case),outside(batteries),unbroken(case)].
```

and the second-level goal is satisfied. So a postcondition recursion for turn_over_case succeeds trivially. The third level of **means_ends** recursion binds its third argument (answer operator list) to **[turn_over_case]**, and its fourth argument (final state) to the just-mentioned state.

8. We return to the second level, and can complete its job with third argument (answer operator list)

```
[disassemble_case,turn_over_case].
```

so we can bind the fourth argument (final state) to the same state list as previously.

9. We finally return to the first level, where were have satisfied the preconditions of replace_batteries. We now figure out the result of replace_batteries. The deletepostcondition is

```
[outside(batteries),defective(batteries)].
```

and the addpostcondition is

```
[inside(batteries),ok(batteries)].
```

so the new state is

```
[inside(batteries),ok(batteries),open(case),
  closed(top),ok(light),unbroken(case)]
```

10. We now must determine how to go from this new state to the original (first-level) goal of

```
[ok(batteries),closed(case),closed(top)].
```

We recursively call **means_ends** to do this. The only difference is **[closed(case)]**, for which the recommended operator is assemble_case. The preconditions of assemble_case are satisfied by the last-mentioned state, so we just work out the postconditions of the operator. This gives the new state

```
[closed(case),inside(batteries),ok(batteries),
  closed(top),ok(light),unbroken(case)]
```

This state matches the goal, so no postcondition recursion is necessary at the second level. So the postcondition recursion for replace_batteries (the top level) finishes with third argument (operator list) being **[assemble_case]** and fourth argument (new state) the preceding state.

11. To summarize, we have for the replace_batteries operator a precondition-satisfying operator list **[disassemble_case,turn_over_case]** and a postcondition-handling operator list of **[assemble_case]**. So the complete operator list for this problem is

```
[disassemble_case,turn_over_case,replace_batteries,assemble_case].
```

and the final state is what we just found,

```
[closed(case),inside(batteries),ok(batteries),
  closed(top),ok(light),unbroken(case)]
```

You can see that means-ends analysis is a little tricky, but it all makes sense when you follow through the steps carefully. Curiously, the operator sequence found seems "obvious" to people. This is another one of those situations mentioned at the beginning of Chapter 2: things that seem obvious at first, but aren't so obvious when you study them. Getting computers to understand language and interpret pictures are also surprisingly difficult problems.

It's interesting to see what happens if for step 3 we try smash_case instead of disassemble_case to get the case open. The smash_case operator doesn't have any prerequisites, but it has the serious postcondition (actually, addpostcondition) that the case is broken. None of the operators can replace a **broken(case)** fact by the **unbroken(case)** we need for the goal state, so further progress is impossible. So if we ever selected smash_case, we would fail and need to backtrack.

Means-ends versus other search methods

The flashlight example illustrates the advantages of means-ends analysis over the classical search methods of Chapters 9 and 10. Consider what those earlier methods would do on the same problem. They would have considered every action possible in the starting state, then every next action, and so on until they reached the goal. Such a search is basically "local" in not having an overall plan of attack, just moving like an ant one step at a time. Heuristics, evaluation functions, and cost functions are just tools to make better local decisions about which operators are most likely to lead to the goal.

Means-ends analysis, on the other hand, reasons top-down from abstractions, gradually filling in details of a solution. It's "global" since it always has an overall idea of what it's trying to do. It uses recursion to decompose a problem in an intelligent way, a way more like how people solve problems. It does require that you identify explicit and complete preconditions and postconditions for each operator; when this is impossible, means-ends won't work.

Another advantage of means-ends analysis is its explainability. Search results in a list of operators to apply in the real world. When this list is long, it's hard to understand and follow, leading to mistakes. But if you can group ("chunk") operators as means-ends analysis does, like saying that a sequence of four operators accomplishes some single overall goal, then a long string is more comprehensible. We'll talk more about explanations in Chapter l5.

Modeling real-word uncertainty (*)

Means-ends analysis plans a solution to (operator sequence for) a search problem; actually using its solution in the real world is another thing altogether. Difficulties occur when assumptions on which the plan is based are violated. For instance, you may assume the flashlight batteries are defective, but discover when you've replaced them that it really was the light that was defective. Or while you were assembling the case, the batteries may fall out. With cases (pun intended) like this we don't need to start over, but we can replan from the intermediate state; that is, do a new means-ends analysis with this intermediate state as the starting state, and goal the same. Do this anytime you find inconsistency between the real-world state and the state anticipated by planning at some point. So even though means-ends analysis is top-down reasoning from abstract goals, you can handle mistakes and imperfections in the world.

Procedural nets (*)

All the search methods in this and the last two chapters find a sequence of states or operators to solve a problem, on the assumption you can do only one action at a time. But this may be false: you may have a multiarm robot, or you may have several people to help you, or actions not requiring supervision can be started simultaneously. As we discussed in Section 10.12, concurrency can speed up search.

Concurrency in an operator sequence can be expressed with the *PERT chart*, a kind of lattice often used in operations research. Figure 11-8 gives an example, one way of replacing both the batteries and light in a flashlight. Each box represents an operator; an operator to the right of another must wait for the first to finish, and vertically parallel operator sequences can be done concurrently. Much work in operations research has studied good ways to design and use such PERT charts efficiently.

PERT charts are often used in artificial intelligence under the name of *procedural nets*. The algorithms used are similar to means-ends analysis but result in a procedural net, by noting multiple differences between a state and the goal that can be eliminated simultaneously by parallel activities. But a major headache with concurrency is possible conflict between simultaneous activities. The postconditions of one concurrent operator might undo the preconditions for another, for instance, requiring that the operators be done in sequence instead of in parallel. So several different conflict-handling approaches have been proposed for procedural nets. They're too complicated to explain in detail here, but involve intersection checks on preconditions and postconditions of operators appearing in parallel; when conflicts are found, the operators are rearranged into a sequence. This may involve removing part or all of the concurrency, or finding a new concurrency. A rule-based system can summarize such fixes.

.SH Keywords:

means-ends analysis
difference table
recursive decomposition
double recursion
hierarchical reasoning
abstraction
planning
partial state description
procedural net

Exercises

11-1. (A) Newspapers often run *household hints* like:

> To remove egg from plastic, use grapefruit juice.
> Wrap sandwiches in lettuce to keep them fresher.
> To make a cauldron bubble faster, use some eye of newt.

What part of means-ends analysis do these correspond to?

11-2. (E) As described, means-ends analysis decomposes a problem into three parts. Explain why a useful variant might be to decompose a problem into five parts. Generally speaking, how could such a variant be implemented in Prolog?

11-3. You are a character in a British murder mystery. You have endured enough of Sir Reginald Finch-Stratton. Not only is he an unbearable egotist, but his bad advice has caused you to lose your life savings on unsound business ventures. Now you hear of his sordid liaison with your daughter, an affair that has left her a broken woman. Clearly the man deserves to die.

(a) Set this up for the means-ends analysis program. Assume his room is next to yours and he keeps the door locked. Assume the operators are to stab him with a knife, to pick up the knife, to hide the knife, to go from a room to the corridor or vice versa, to knock on his door, and to wait until the corridor is empty. Needless to say, you don't want to be caught in this foul deed, so you don't want anyone to see you or to find the murder weapon. Assume in the starting state that you are in your room, there are people in the corridor, you are not holding the murder weapon (knife), Sir Reginald is alive, Sir Reginald is in his room, and the door is locked.

(b) Trace through informally the operation of means-ends analysis on this problem. What is the final operator sequence found?

(c) Some of the things assumed in this problem are hard to know in advance, like whether there will be people in the corridor or not. How in general can means-ends analysis be modified to handle such randomness in facts?

11-4. (R,A,P) In many flashlights the top consists of three parts instead of the two assumed in this chapter: a holder, a transparent plate, and a light bulb that screws into the holder underneath the plate. Modify the flashlight repair program to handle flashlights with these three parts, in which only the light is ever defective. You'll need to define new operators **remove_plate** and **replace_plate**; the plate must be removed before the light can be removed, and the plate must be replaced before the top can be assembled. You'll also need some new predicates to include in states.

Try it out on the same repair situation given in the chapter.

11-5. (P) Consider using the flashlight repair program for the situation in which both the light and the batteries are defective.

(a) What does the program in this chapter give as an answer operator list? (Use a computer if you have one.)

(b) The order of the **recommended** rules can make a lot of difference. Suppose you move the second **recommended** rule (the one about replacing the light) to the end of the others. What answer do you get now? Why?

(c) Suppose you instead move the first **recommended** rule (the one about replacing the batteries) to the end of the end of the others. What answer do you get now? Why?

11-6. (A,E) Auto repair manuals often suggest laying out the parts removed in order in a line on a flat surface. Assuming that people do auto repair by something close to means-ends analysis, explain in artificial-intelligence and computer-science terms why this way of laying out is a good idea.

11-7. (A) So far, preconditions and postconditions have been lists of predicate expressions "and"ed together. This seems limiting. Suppose some operator has two alternative sets of preconditions, either of which can be satisfied to make the operator usable. Give an easy way to represent such situations when using the mean-ends analysis program in this chapter.

11-8. (E) The means-ends analysis program of this chapter works in a depth-first way: it figures out how to satisfy the preconditions of a recommended operator, possibly requiring more recursions, and then it figures out how to go from the result of the operator to the goal, possibly requiring more recursions. So it's depth-first in style, with a branching factor of 0, 1, or 2 at each node (each invocation of the **means_ends** predicate). But this is not strictly necessary.

(a) Explain how a breadth-first means-ends analysis might work.

(b) Why do you think we used the depth-first approach in this chapter?

11-9. (E) Discuss the application of means-ends analysis to improve the user-friendliness of using networks of computers. Give an example.

11-10. (E) Discuss the similarities and differences between means-ends analysis and the top-down parsing of strings of words as discussed at the end of Chapter 6.

11-11. (H) Machines now exist to synthesize pieces of DNA (deoxyribonucleic acid) under microprocessor control. Chemical reactions take place in a single test tube. The following actions are possible:

1. Add plastic beads to the test tube.
2. Add one of four similar "bases" to the tube (symbolized by A+X, C+X, G+X, and T+X).
3. Add a chemical that strips the X off bases (for instance, A+X becomes A)
4. Add a chemical that binds the left ends of bases to beads.
5. Neutralize the chemical that binds the left ends of bases to beads.
6. Add a chemical that detaches the left ends of bases (or strings of bases) from beads.

The following conditions apply:

1. The right end of a base will attach to the left end of another base, but only if there is no X on the right end of the first base.

2. If an X is removed from a base and the base is not used immediately, it will decompose

into a different (useless) chemical.

For this problem, use means-ends analysis to find a plan to synthesize the chemical A+C+G (and only that chemical, no other combinations). That is, accomplish both the operations "attach C to right end of A" and "attach G to right end of C".

Go to book index

Abstraction of facts

Abstractions of search problems were considered in the last chapter. We'll now look at abstractions on groups of facts: *frames*. Frames are invaluable when you've got lots of facts because they organize them. We'll discuss the different kinds of frames, the components of frames, and new issues in inheritance with frames.

Partitioning facts

Frames--sometimes called *classes*, sometimes *prototypes*, and sometimes *structured-object descriptions* (which doesn't necessarily mean "objects" with physical presence)--partition facts in an artificial-intelligence system. Partitioning of large programs into subroutines and procedures is very familiar in computer science; Section 6.6 mentioned partitioning rules into groups related by topic. But we can also partition data (facts). The computer language Smalltalk exploits extensive fact partitioning. The usual way is to group together facts about the same thing (or *object*).

Prolog dialects usually already have built-in fact partitioning in their indexing together of the facts with the same predicate name. But frames usually work differently: they group facts with the same argument values. So these facts

```
a_kind_of(enterprise,carrier).
a_kind_of(vinson,carrier).
location(enterprise,san_diego).
location(vinson,san_francisco).
color(enterprise,gray).
color(vinson,gray).
east_of(vinson,enterprise)
```

might make two frames (partitions): one of facts about the Enterprise (the first, third, fifth, and seventh facts) and one of facts about the Vinson (the second, fourth, sixth, and seventh facts). With the built-in Prolog indexing by predicate name, there would be four implicit partitions: the two **a_kind_of** facts, the two **location** facts, the two **color** facts, and the **east_of** fact. The first way seems more like how people organize facts in their heads.

Frames, like semantic networks, are intended for two-argument predicates. This includes the Chapter 2 categories of relationship predicates (in which both arguments are objects), and property predicates (in which the first argument is an object, the second argument a property of that object). (If you have more than two arguments to your facts, you must convert them to relationships and properties somehow.) Relationship-predicate facts are generally stored only with the first-argument frame; then for the second-argument frame, a different "reverse" predicate name is used. For instance, for **a_kind_of(X,Y)**, **a_generalization_of(Y,X)** is the reverse; for **part_of(X,Y)**, **contains(Y,X)** is the reverse.

Frames and slots

A frame is more than a collection of facts; it is an abstraction in its own right. Often we feel that certain facts are the only essential ones about some object, the ones giving descriptive *completeness*. For instance, for a ship the name, identification number, type of ship, nationality, tonnage, and location might be the important features. We should make sure we always put those facts in a frame for a ship.

There's a problem, however: we may want to describe sets of objects with a frame, not just a single object. Why? We might want to generalize about certain groups or classes of objects, identifying properties each group shares. Then we can't always fill in all the essential facts in the frame since different objects may have different facts true. For instance, a frame for American ships: each ship has a different identification number, so we can't assert a single identification-number fact for the entire frame.

But identification number is still a *potential* fact for the frame. The terminology is that frames have *slots* that can be *unfilled* or *filled*; filled slots represent facts. For instance, ships vary in location, so frames for "ship" and "carrier" should leave the "location" slot unfilled; but the "transport medium" slot of ships should always be filled with the word "water". Some slots are so important to fill that their frames don't make sense otherwise. Such slots are *definitional*, part of the definition of the frame. For instance, the "nationality" slot of the frame for "American ships" must be filled in with the value "American", because that's what the frame is all about. Many pairs of frames having **a_kind_of** relationships have such slots, which came about when the more general frame was restricted in some slot to create the more specific frame.

Inference rules can fill unfilled slots, especially inheritance rules (Section 4.9). That is, to fill a slot in a frame F, find a related frame F2, take the value for the same-named slot in F2 provided the slot inherits across the relationship between F and F2. For example, if all carriers are gray, that color fact can be stored with the "carrier" frame and inherited via **a_kind_of** to the frames for the carriers "Enterprise" and "Vinson". Inheritance can be overridden by putting explicit values into normally-unfilled slots, stating the exceptions to general principles. For instance, the few carriers that are not gray can have an explicit color value in their "color" slot.

Slots qualifying other slots

Slots may have more than a value attached to them: they can have information that explains or qualifies that value. Slots with numeric values can have associated information about the units of measurement and its accuracy. Slots with nonnumeric values can have a format (a formal description of what the values look like). Slots representing real-world data can have an associated location, time, and observer. Slots can have default or "usual" values. Slots may have associated sets of permissible values, given by a list if the number of such values is finite, or by a range if values are numeric. Slots may also have sets of "unusual" values that should generate a warning if seen.

For instance, a slot for the current latitude of a ship can have as explanatory or qualifying information:

> --an indication that the format is a one-digit or two-digit number followed by the letter N or S;
>
> --the units "degrees";
>
> --the time a ship was at that position;
>
> --who reported it;
>
> --a note that the number must be less than 91;
>
> --a note that it's unusual for the number to be more than 60;
>
> --a default value that is the latitude of Norfolk (a good default for U. S. Navy ships, anyway).

Such *slot qualification* information can be considered as slots itself, and we'll call them *qualifying slots*. They can inherit like regular slots. For instance, the restriction that the latitude be less than 91 holds for anything on the surface of the earth, not just a ship, and so could inherit from the corresponding information in a more general frame like "vehicle_on_earth" or "physical_object_on_earth". Notice that qualifying slots generally inherit from a higher-generalization frame than do the slots they describe, when both inherit.

Frames with components

Frames can represent things with components. Often just listing component names is insufficiently descriptive. Instead we should have a separate frame for each component, and **part_of** slots pointing to them. This has several advantages:

> --We can distinguish properties of components not shared by the whole (inheritance can always be used otherwise).
>
> --We can describe relationships of the components to one another, like the relative location of parts of a physical object or the relative time of subevents of an action. Frames for which components can be put in a linear sequence are called *scripts*.
>
> --We can indicate more easily how many times component frames occur in relation to the

main frame (*cardinality* information, which can be expressed as a qualifying slot). For instance, a wheel is part of a car, and cars have four of them, each with their own properties; and a sunroof is an optional part of a car, occurring either zero or one times in every car.

--We only need identify the most important, "top-level" components in our main frame. The frames for these components can describe their subparts, and so on.

Frames as forms: memos

Frames have many applications to management of paperwork, because frames are a lot like forms. For instance, consider memos used for communication in organizations. Figure 12-1 shows some memo frames linked in an **a_kind_of** lattice. All memo frames have certain slots: author, intended readers (addressees), date written, subject, type (formal or informal) and text. Often the first four are even written on the paper to look like slots, with the symbols TO, FROM, DATE, and RE followed by colons.

The general memo frame has many (not necessarily exclusive) sub-frames, each linked by **a_kind_of(X,memo)** facts. Examples are a frame for all the memos written by Ann, a frame for all the memos received by Ann, and a frame for all memos about the budget. These last two have, among others, a sub-frame of their own: all memos received by Ann about the budget. And this might happen to have another sub-frame, for a specific memo Ann got on this subject the other day. Of course, there are many other memo sub-frames. Just considering sub-frames restricting the values in the "subject" slot, there are "policy" memos, "announcement" memos, and "personal" memos. Considering sub-frames restricting the "date" slot, there are memos of critical and short-term importance, memos of immediate but long-term importance, and memos of only long-term importance. Finally, note that the "memo" frame itself is a sub-frame of a "business form" frame.

Since memo components appear in sequence down a page, their frames are scripts. But organizations do differ in these orders, so the component sequence is inherited from frames specific to each organization.

The author, addressee, and text slots shown are special to memos because they must be filled in for every specific memo *instance* (that is, real-world memo), though not for every memo frame (which can represent classes of memos), with new slots added at various levels in the lattice of memo frames. This is a special type of "cardinality" information for those slots, which can be expressed in qualifying slots.

Slot inheritance

All memos have a text slot, but this isn't filled in unless we're talking about a specific real-world memo; every memo's text is different, obviously. But you must agree that since the general "memo" frame has a "text" slot, the "memos to Ann" frame must have a "text" slot too. This is inheritance, but of a fundamentally different sort than any in this book so far: it's inheritance of the mere *concept* of a slot, and not the value in a slot. We'll call this *slot inheritance*, and the regular kind *value inheritance*. In one sense, slot inheritance is more limited than value inheritance in that it usually only works with the **a_kind_of** relationship. That is, situations when some frame F has a slot S, then anything that is "a kind of" F also has slot S. But in another sense, slot inheritance is more general than value inheritance because it applies higher up in frame hierarchies, to slots that don't yet have values filled in.

Usually slot inheritance tells us most of the slots that must be in a sub-frame given the slots in a frame. But some slots may be unique to the sub-frame. For instance, carrier ships have airplanes, whereas ships in general don't. So slots referring to airplanes, like those indicating number and types, are unique to the "carrier" frame. But slots in a frame can't "disappear" in a sub-frame.

Part-kind inheritance

Besides value and slot inheritance there is a third fundamental kind of inheritance: part-kind inheritance. It occurs in the interaction between **a_kind_of** and **part_of** (see the example in Figure 12-2). A ship has decks, a hull, and a propulsion system. So the ships Enterprise and Vinson have those things. But the Enterprise's hull is *different* from the Vinson's hull. It may have different properties: it may be damaged while the Vinson's hull or the hull of ships in general isn't. Some kind of inheritance is happening from the "hull" frame, but it's not inheritance of values or slots because "hull" isn't either a value or slot but a frame itself. We'll call this *part-kind inheritance*. Formally, it's the inheritance of a component FC of some frame F of which frame FK is "a kind of"; the inherited frame becomes a component of FK, inheriting slots and values from both FK and FC. Part-kind inheritance means you don't need to define every frame in advance for which you want to assert slot values: some frames must exist because others exist. Part-kind inheritance is often signaled in English by possessives, like "Enterprise's" in the phrase "the Enterprise's hull".

Extensions versus intensions

Philosophers make an important distinction between abstract concepts (what they call *intensions* or *meanings* of concepts) and real-world things that frames correspond to (what they call *extensions*). (Don't confuse this first word with the more common but differently-spelled one "intention".) For instance, the intension of "carrier" is an abstract specification of what it means for something to be a carrier. One extension of "carrier" is the set of all currently existing carriers, another the set of all carriers that existed a year ago, another the set of all carriers ever built, and another the set of all carriers that will exist ten years from now (the "world" of the extension may be hypothetical).

Extensions and intensions generally require separate frames, linked by "extension" and "intension" pointers to one another, because the slots can be filled differently. For instance, it's technically incorrect to say that the Enterprise is at 14N42W, only that the extension of the Enterprise for the current instant of time has a location value of 14N42W. So only extensions of things can have a location. In general, extensions can have statistics while intensions cannot. So if you want to fill a slot with something that's usually true or true on the average, you need an extension frame. For instance, if you want to say "most tankers are Liberian", you're making a statement about the nationality statistics of some extension of the abstract concept of "tankers".

Properties of extensions usually inherit from their intensions: an abstract concept constrains the examples of it. For instance, the fact that ships float on water by definition (i.e., as a property of their intension) means that existing ships (i.e., an extension) float on water. But statistical properties of extensions rarely inherit in any direction. For instance, the average length of a memo in some organization isn't likely to be the average length of a policy memo in that organization. Weak inferences are sometimes possible between sets and their subsets, if the condition of statistical independence holds; sampling theory from statistics can help us recognize such situations.

Procedural attachment

Inferences rules (or inference procedures) can be values in slots. Usually they appear in qualifying slots, and represent a way to fill the value in the main slot from other accessible values. This is *procedural attachment*, and the procedures are sometimes called *if-needed rules*. Rules can inherit just like other slot values, so we can specify a rule for many frames with a single value entry.

Extensive or exclusive use of procedural attachment in an artificial-intelligence system leads to a whole new style of programming, *object-oriented programming*. We mentioned it back in Section 6.9, as one interpretation of parallelism in rule-based systems. Object-oriented programming is especially useful for writing simulations. The simulation is divided into *objects*, each with its own frame and rules running independently. This is a sort of opposite to the *procedure-oriented programming* using Prolog that is emphasized in this book, for which procedures (rules) call on data. With object-oriented programming, data (in frames) calls on procedures. As such, object-oriented programming requires a whole different mindset or programming philosophy than procedure-oriented programming, and it's hard to intermingle it with the other things we discuss in this book. If you're interested, take a look at the Smalltalk literature.

Frames in Prolog

Frames are best implemented in Prolog with software modules where each module has its own local facts, rules, and indexing scheme. Unfortunately, only some of the currently available Prolog dialects provide true modules. But some of the meaning of frames can be captured by identifying them with files, making each frame a separate file.

Filled slots in a frame can just be facts in the file with the usual predicate names. Unfilled slots can be represented by facts of the form

```
slot(<object>,<slot-name>).
```

assuming **slot** is not a slot name itself. This special predicate can inherit by slot inheritance from that object is a kind of.

Qualifying information (like units, bounds, and measurement accuracy) can be specified as special property facts where the predicate name is the name of the qualifying slot, the first argument is name of the slot qualified, and the second argument is the value in the qualifying slot. For instance:

```
units(length,meters).
```

Ordering between parts can be modeled by additional relationship facts besides those designating the parts themselves. For instance:

```
beneath(engine_room,flight_deck,carrier).
```

which says that the engine room part of a carrier is beneath the flight deck part.

Example of a frame lattice

Here are some interrelated frames about cars; Figure 12-3 shows their relationships. To make inference rules simpler, we use triples to represent slot values instead of the usual two-argument predicates. The syntax is **value(<object>,<slot>,<value>)** which means that the **<slot>** of the **<object>** has the **<value>**.

Physical-object frame:

```
slot(physical_object,weight).
slot(physical_object,name).
slot(physical_object,use).


units(physical_object,weight,kilograms).
```

Vehicle frame:

```
value(vehicle,a_kind_of,physical_object).
value(vehicle,use,transportation).
value(vehicle,has_a_part,propulsion_system).


slot(vehicle,owner).
slot(vehicle,dealers).
slot(vehicle,year).
slot(vehicle,age).
slot(vehicle,propulsion_method).
```

```
units(vehicle,age,years).
units(vehicle,year,years).
```

Car frame:

```
value(car,a_kind_of,vehicle).
value(car,propulsion_method,internal_combustion_engine).
value(car,has_a_part,electrical_system).
value(car,extension,cars_on_road).


slot(car,make).
slot(car,model).


possible_values(car,make,
  [gm,ford,chrysler,amc,vw,toyota,nissan,bmw]).
```

Electrical-system frame:

```
value(electrical_system,part_of,car).
value(electrical_system,has_a_part,battery).
value(electrical_system,has_a_part,starter).
```

VW-Rabbit frame:

```
value(vw_rabbit,a_kind_of,car).
value(vw_rabbit,make,vw).
value(vw_rabbit,model,rabbit).
```

Cars-on-the-road-now frame:

```
value(cars_on_road,intension,car).


statistic(cars_on_road,mean,age,6.4).
```

Joe's-VW-Rabbit frame:

```
value(joes_rabbit,a_kind_of,vw_rabbit).
value(joes_rabbit,extension,joes_rabbit_now).
value(joes_rabbit,owner,joe).
value(joes_rabbit,year,1976).
```

Joe's-VW-Rabbit-now frame:

```
value(joes_rabbit_now,subset,cars_on_road).
value(joes_rabbit_now,intension,joes_rabbit).


statistic(joes_rabbit_now,size,none,1).
```

Joe's-VW-Rabbit-battery frame:

```
value(joes_rabbits_battery,extension,joes_rabbits_battery_now).
value(joes_rabbits_battery,part_of,joes_rabbit).
```

Joe's-VW-Rabbit-battery-now frame:

```
value(joes_rabbits_battery_now,intension,joes_rabbits_battery).
value(joes_rabbits_battery_now,contained_in,joes_rabbit_now).
value(joes_rabbits_battery_now,status,dead).
```

Now for some inference rules applying to these frames. We first define a single general-purpose value-inheritance rule, that can do inheritance for arbitrary predicate names. This makes inheritance implementation a lot simpler since we don't have to write a separate rule for each predicate that inherits. Technically, we've implemented something close to a *second-order logic*, something that reasons about predicate names as well as arguments, as we'll explain more in Chapter 14.

```
has_value(Object,Slot,V) :- value(Object,Slot,V), !.
has_value(Object,Slot,V) :- inherits(Slot,Relation),
  value(Object,Relation,Superconcept),
  has_value(Superconcept,Slot,V).


inherits(S,a_kind_of) :-
  member(S,[use,
   propulsion_method,dealers,year,age,make,model]).
inherits(S,part_of) :-
  member(S,[owner,dealers,year,age,make,model]).
```

Slot inheritance must be defined separately:

```
has_slot(Object,Slot) :- slot(Object,Slot), !.
has_slot(Object,Slot) :- value(Object,a_kind_of,Superconcept),
 has_slot(Superconcept,Slot).
```

Qualifying slot values also inherit downward. For instance, units of a slot:

```
has_units(Object,Slot,U) :- units(Object,Slot,U), !.
has_units(Object,Slot,U) :- value(Object,a_kind_of,Superconcept),
 has_units(Superconcept,Slot,U).
```

Here's a rule that says to get a slot value of an extension, get the corresponding slot value (possibly inherited) in the intension.

```
has_value(Extension,Slot,V) :- value(Extension,intension,I),
  has_value(I,Slot,V).


has_slot(Object,Slot) :- value(Object,intension,I), has_slot(I,Slot).
```

As we said, statistics can sometimes inherit from one extension to another, if an "independence" condition holds:

```
statistic(Extension,Statname,Slot,Value) :-
  has_value(Extension,subset,Bigextension),
  independent(Extension,Bigextension),
  statistic(Extension,Statname,Slot,Value).
```

And the subset relationship holds between two extensions whenever their intensions have an **a_kind_of** relationship:

```
has_value(Extension,subset,Bigextension) :-
  value(Extension,intension,I), has_value(I,a_kind_of,BigI),
  value(BigI,extension,Bigextension).
```

Also, some redundant slots may have their values defined in terms of other slot values. The **has_a_part** relationship is just the opposite of the **part_of** relationship:

```
value(X,has_a_part,Y) :- value(Y,part_of,X).
value(X,part_of,Y) :- value(Y,has_a_part,X).
```

And the age slot value can be defined from the year slot value:

```
value(Object,age,A) :- value(Object,year,Y), current_year(Y2),
  A is Y2 - Y.


current_year(1987).
```

Here are some sample queries run with the preceding facts and rules:

```
?- has_value(joes_rabbit_now,use,U).
U=transportation


?- has_value(joes_rabbits_battery_now,age,A).
A=11


?- has_slot(joes_rabbit_now,name).
yes


?- has_value(joes_rabbit_now,subset,X).
X=cars_on_road
```

Expectations from slots

An important applications of frames is to modeling and reconstruction of stereotypical situations in the world from incomplete knowledge. Empty slots in a frame have "expectations" about what should fill them: from inheritance, from qualifying-slot information (possible values and permissible values), and from extension statistics. Consider purchasing of equipment for a bureaucratic organization, which usually involves many steps and many details; if we know some of the details (slot values) of the purchase, then other details (slot values) are often obvious. For instance, an arriving order was probably ordered six to three weeks ago; orders from accounting-supplies companies are for the Accounting department; orders that come by express mail are probably for management and should be delivered immediately.

Frames for natural language understanding (*)

People must exploit expectations to understand natural languages, because speakers and writers try to avoid wordiness. So it's not surprising that frames are very helpful for natural-language understanding by computers, for the *semantics* or meaning-assignment subarea (as opposed to the *syntax* or parsing subarea we discussed in Section 6.12). That is, with a good frame representation we can efficiently capture the meaning of some natural-language sentences, so as to answer questions about it.

Usually the goal in interpretation of a sentence or sentences is to get a set of interrelated frames, in which each frame represents a verb or noun and its associated modifiers. Verbs and nouns are frames, and modifiers are slots or have something to do with slots. So for instance the sentence

> "Yesterday we sent headquarters by express mail the budget memo that Tom drew up for Ann on 6/12".

can be represented by three frames as in Figure 12-4: an instance or sub-frame of a "sending" frame, an instance of a "memo" frame (like those in Figure 12-1), and an instance of a "drawing-up" frame. These three are linked by the uses of their names as slot values. Some implications of the sentence are also filled in; for instance, the person a memo is drawn up for is assumed an addressee of the memo. Note that if we know more about the sending, drawing up, or the memo itself from other sentences, we could fill in additional slots in the frames without necessarily requiring more frames.

Filling in frames the right way to capture the meaning of a sentence can involve search. To be sure, the parse of the sentence (see Section 6.12) helps us considerably by identifying the grammatical categories of each word. But there are many ambiguities that can't be resolved by a parse. For instance, compare the previous sentence to:

> "Yesterday we sent several times by 4 P.M. the budget memo that Tom drew up for practice on the plane".

Here we have "several times" instead of "headquarters", "by 4 P.M." instead of "by express mail", "for practice" instead of "for Ann", and "on the plane" instead of "on 6/12". In all cases we've substituted something similar grammatically. But the functions of the substitutions are different: "several times" describes the style of the sending, not the place we sent to; "by 4 P.M." is a time limit, not how we sent; "for practice" is a purpose, not a beneficiary; and "on the plane" is a location, not a time. We must figure these things out making guesses about words, drawing from their possible meanings, and checking the resulting interpretation of the whole sentence for reasonableness.

Multiple inheritance (*)

We haven't discussed how to handle "multiple inheritance" paths for some slot of a frame. For instance, a policy memo from your boss is simultaneously a memo from your boss and a policy memo. Things are fine if only one path provides an inheritance for a slot, or if the paths all agree on some value. But if different paths give different values for a slot, we must do something. If we can assign priorities to paths, we can take the value from the one with highest priority. Or we can compromise or find a middle ground between different values. Or we can decide certain values are wrong. The next section presents an example application.

A multiple inheritance example: custom operating systems (*)

Frames are valuable in managing large software systems. As an example, take the operating system of a computer, the top-level program that runs other programs on your orders. Operating systems are standardized for an average computer user, with not too many things you can adjust. Users would prefer operating systems more custom-fitted to their needs.

One way is by a hierarchy of user models. Think of each user model as a frame holding information about defaults peculiar to a user or class of users, defaults about how they need or prefer to use particular programs and facilities. Some possible slots are:

--default terminal setting parameters;

--storage and time allocations (different programs and facilities have quite different requirements);

--the project for which this computer time should be charged;

--default non-file parameters of the program or facility itself (like for a printer, whether output is double-spaced);

--protection information for any input and output files (to prevent reading nonmeaningful files or overwriting valuable files);

--additional character strings (extensions) to be automatically added to the names of input and output files (like "pro" for all Prolog programs);

--interrupt-condition handling (like what to do on arithmetic overflow);

--common misspellings (so they can be recognized);

--common bugs (so they can be caught before damage is done);

--pointers to documentation;

--names of default editors, document handlers, and programming languages (so the user can just type short words like "edit" and the operating system will know what they mean).

The interesting thing about this application is the three independent inheritance hierarchies (see Figure 12-5). First, there's a hierarchy of user classes: frames for Tom, frames for people in Tom's project group above that, and frames for everybody using the operating system above that. This hierarchy may be a lattice, because Tom may belong to more than one project group, each with different associated projects. Second, there's a hierarchy on programs and facilities of the operating system: a frame for Tom using the Prolog interpreter, and a frame above that for Tom using any programming language, and a frame above that for Tom doing anything under the operating system. This hierarchy could be a lattice too, since for example a Prolog interpreter is both a Prolog-related facility and an interpreter. Third, there's a hierarchy on time: Tom's use of the Prolog interpreter today is below a frame for Tom's use of the Prolog interpreter anytime.

So there are at least three independent dimensions of frame generalization for the frame representing Tom using the Prolog interpreter today, the bottom frame in Figure 12-5. We can represent them by a three-dimensional lattice (which Figure 12-5 tries to suggest) in which the first (user-class) dimension runs northwest, the second (facility-class) dimension runs northeast, and the third (time) dimension runs straight north. Though each of the three hierarchies here is a linear sequence, they form a lattice when put together, since there are many routes between the top frame and the bottom frame.

The big problem is thus multiple inheritance: there are three different directions to reason. This doesn't affect slot inheritance for which nothing conflicts, but it is a serious problem for value inheritance. One approach is to give the **a_kind_of** link for each hierarchy a different name, like **a_kind_of_1**, **a_kind_of_2**, and **a_kind_of_3**. Then for every slot, designate one of these as preferred for inheritance, and store this in a qualifying slot of the original slot. For instance:

> --inherit a value for the "sponsoring projects" slot via the user-class hierarchy, since projects are composed of people;
>
> --inherit terminal settings from the program and facility class hierarchy;
>
> --inherit program storage allocation from the time hierarchy, the typical allocation in the past.

This designated-predicate idea runs into trouble when a user wants to override defaults obtained by inheritance. For instance, suppose user Tom wants to override default terminal settings for the Prolog interpreter, obtained from traveling up the program-class hierarchy to "Tom on languages today", and suppose Tom wants this override for all time (that is, Prolog programs might benefit from consistently unconventional settings). So he should place the overriding settings in the frame above him in the time dimension ("Tom on Prolog anytime") representing all his uses of the Prolog interpreter for all time. But now we must prevent inheritance along the program dimension. A fix is to store for each slot a sequence of inheritance predicates that should be tried to find the value of the slot--for this example perhaps the sequence of the time dimension, the program dimension, and the user-class dimension. Each slot can have an associated sequence, and this can inherit.

This approach to inheritance is general, but sometimes unwieldy. For particular slots we may be able to do something simpler. For instance, we may be absolutely certain that values won't conflict because of the way we've built the frames hierarchy. Or for slots with numeric values, we can apply a numeric function to the values inherited along different dimensions to get a "compromise" number. For example, for a storage allocation slot, we could take the maximum of the storage allocation values found along the three dimensions, because each dimension represents a different requirement that must be met. For the time allocation slot (the amount of time given to a running program before asking the user if they wanted to continue), we could take the average of values, since each number is just a rough guess about an intuitive parameter. This idea of a compromise value applies to nonnumeric slots too. For example, if Tom's project group insists on putting its name at the end of every file created by that group, while a Prolog compiler insists on putting "procomp" at the end of the name of every file compiled from Prolog code, you can compromise by putting first the project name, then "procomp". Such specialized multiple-inheritance strategies (not to be confused with "conflict-resolution" strategies for rule-based systems) can be flagged by a special "multiple-inheritance method" qualifying slot. And this slot can inherit too.

.SH Keywords:

object-based representation
frame
class
slot
filling a slot
definitional slots
qualifying slots
script
value inheritance
slot inheritance
part-kind inheritance
intension
extension
procedural attachment
semantics
multiple inheritance
frame lattice

Exercises

12-1. Consider a frame representing any class meeting at any university (that is, a generalized class hour).

(a) Give four example slots that are always filled with values and what those filled values are.

(b) Give a superconcept frame (a frame that this frame is a kind of) and a subconcept frame (a frame that is a kind of this frame).

(c) Give a slot that inherits from your superconcept frame and one that does not.

(d) Describe a script associated with a class meeting frame.

12-2. (E) People dealing with bureaucracies are constantly filling in the same information on forms over and over again. Discuss the use of inheritance to lighten this burden. In particular, explain how inheritance could have a concrete physical meaning.

12-3. (R,A) Consider a frame representing any purchase order (a form ordering the buying of something).

(a) Give an example of value inheritance from a value in this frame to a value in some other frame.

(b) Consider the "units" slot associated with some slot S in the purchase order frame (so if S is the cost slot, units would be "dollars"). Suppose S inherits its value from the same slot S of some other frame P2. What does this tell you about from where the "units" slot inherits?

12-4. (A) Different slots in a frame system require different qualifying slots to accompany them. But every time we use a slot with a particular name, we want it to have the same qualifying slots, and the same values in those qualifying slots that have values filled in. We can't always just inherit this information from a higher (more general **a_kind_of**) frame because, say, two different uses of a "length" slot might be in frames having no common ancestor having a "length" slot. (This is especially a problem when different people are building different frames for a big system.) Suggest an elegant way to handle this sort of problem.

12-5. (R,A) Every computer has a manufacturer's name and an ID number. Every computer has a CPU as a component. Consider the frame representing any computer sold by Floodge Manufacturing, Inc.

(a) Give an example of downward value inheritance from this frame.

(b) Give an example of downward slot inheritance from this frame.

(c) Give an example of downward part-kind inheritance involving this frame.

12-6. Contrast the intension and extension of the concept "memo".

12-7. Draw a semantic network representing the following facts. Represent property values by nodes too (though this isn't always a good idea). Represent what these mean, not what they literally say.

> Wrenches are tools.
> Hammers are tools.
> Tools have handles.
> The handle of a hammer is hard.
> Wrenches are hard.
> Most hammers have a steel handle.
> Wrenches are 10 inches long on the average.

12-8. Names of frames often correspond to nouns. For instance, a ship frame corresponds to the meaning of the English word "ship".

(a) What, in frame terminology, does an adjective modifying the noun name of a frame often correspond to? For instance, the adjectives "big", "American", and "merchant" in the description "big American merchant ships".

(b) What, in frame terminology, does a preposition in a prepositional phrase modifying the name of a frame often correspond to? For instance, the prepositions "in" and "at" in the description "ships in the Mediterranean at noon yesterday".

12-9. (A) Consider the following example of reasoning by modus ponens:

> Given: Military organizations are widely dispersed geographically.
> Given: The Navy is a military organization.
> Hence the Navy is widely dispersed geographically.

That conclusion makes sense. But consider:

> Given: Navy organizations are widely dispersed geographically.
> Given: The Naval Postgraduate School is a Navy organization.
> Hence The Naval Postgraduate School is widely dispersed geographically.

That is an incorrect inference assuming the "Given"s are true. Discuss why the two situations are different. (To get full credit, you must show depth of understanding, not just cite a superficial difference.)

12-10. Represent the meaning of the following letter with a group of interlinked frames.

> Dear Fly-by-Nite Software,
>
> I tried your product "Amazing Artificial Intelligence Software" and it doesn't work. I tried Example 5 shown in the manual, and it crashed on an attempt to divide by zero. I next tried Example 7, and the lineprinter printed 100,000 line feeds. Then when I exited your program, I found it had destroyed all my files. I want my money back.
>
> Sincerely,
> Irate Programmer

12-11. Some slots are easier to handle with multiple inheritance than others.

(a) Consider associated with a nonnumeric slot a qualifying slot that lists all the conceivable values that first slot could have. Formulate a multiple inheritance policy for this qualifying slot.

(b) Consider associated with a numeric slot a qualifying slot that lists the maximum and minimum conceivable values for that first slot. Formulate a multiple inheritance policy for this qualifying slot.

12-12. For the user-model hierarchy example, formulate conflict-handling methods for multiple inheritance of the following.

(a) The directory to which output files go.

(b) A bit indicating whether the user should be warned before the operating system executes a command whose side effect is to destroy a file.

(c) The "CPU intensity" of usage, defined as the expected ratio of CPU time to login-connection time.

12-13. (A) Suppose that for our user model frames example we have qualifying slots describing how to resolve multiple inheritance on other slots. These qualifying slots could have qualifying slots themselves explaining how to resolve their own multiple inheritance problems, and these could have qualifying slots themselves, and so on. Why shouldn't we worry about an infinite loop?

12-14. (E) Chapter 7 considered the building of an expert system for appliance repair. Suppose instead of one general expert system we would like a set of expert systems, one for each kind of appliance. Many rules are common to different appliances, but some rules must be deleted, some added and some modified in going from one appliance to another. Discuss how it might be useful to define an appliance's rules as a frame and do inheritance. What slots would be necessary and useful in the frames, and how would you fill in their values?

12-15. (E) Discuss the use of a frames hierarchy to represent a contract negotiation between management and labor. Each frame will represent a particular offer or proposal. Some of these can be grouped together. State what slots can be useful for these frames. Provide a good representation of negotiation so that later analysis can more easily pick out patterns in the style of negotiation such as flexibility.

12-16. (H,E) Reasoning by analogy is a fundamentally different kind of reasoning than any so far considered in this book. It has similarities to inheritance, but it's really something else entirely: it involves four things instead of two. It's easiest to understand in terms of frames and slots. Suppose you want to know about the slots or slot values in some frame D. You could find frames A, B, and C such that the relationship of A to B is very similar to the relationship of C to D, the frame of interest. For instance, A might be a frame representing the circulatory system of the human body, and B a frame representing the medical emergency of a heart attack; then C might represent the cooling system of a car and D the serious malfunction of complete coolant blockage. Using the analogy, we might be able to infer that the "number of previous attacks" in frame B should correspond to a slot "number of previous blockages" in frame D. Furthermore, the value in the "immediate treatment" slot of frame B of "inject anticoagulants into the system" could have an analogy in a slot "immediate treatment" of frame D of "add anti-corrosives". Give a general strategy for finding such analogous slots and analogous slot values in reasoning-by-analogy situations.

Go to book index

Problems with many constraints

Some problems have lots of conditions or *constraints* that must be satisfied. When expressed as Prolog queries, these problems appear as many predicate expressions "and"ed together. Processing these queries can be awfully slow if we don't do it right.

Special techniques are often used for such many-constraint problems. We'll talk about three such ideas in this chapter: rearranging queries to make them easier to solve, smarter ("dependency-based") backtracking, and relaxation (reasoning about possibility lists).

Two examples

Two good examples of many-constraint applications are automatic scheduling and computer vision.

Suppose we want to schedule a class to meet five times a week. Assume meeting times must be on the hour, from 9 A.M. to 4 P.M., and on weekdays. Suppose we have ten students, each of which is taking other already-scheduled classes, and we can't have our class meet when any of those meets. Furthermore, suppose we don't want our class to meet two successive hours on any day, and we don't want our class to meet more than two hours total on any day. Figure 13-1 gives an example situation and an example solution; "occupied" represents times that aren't available because of the other commitments of the ten students, and "class" represents a solution time proposed for our class. (Blank spaces are available times we don't need to use.)

This problem can be stated as a query finding values of five variables **T1**, **T2**, **T3**, **T4**, and **T5**, each value of which is a two-item list **[<day>,<hour>]**, so **[tuesday,3]** means a meeting on Tuesday at 3 P.M. Suppose meeting times of other classes are represented as **occupied([<day>,<hour>])** facts. Then the following query would solve our problem:

```
?- classtime(T1), classtime(T2), classtime(T3),
  classtime(T4), classtime(T5),
  not(occupied(T1)), not(occupied(T2)), not(occupied(T3)),
  not(occupied(T4)), not(occupied(T5)),
  not(two_consecutive_hours([T1,T2,T3,T4,T5]),
  not(three_classes_same_day([T1,T2,T3,T4,T5]).
```

For this we need the following definitions:

```
classtime([Day,Hour]) :-
  member(Day,[monday,tuesday,wednesday,thursday,friday]),
  member(Hour,[9,10,11,12,1,2,3,4]).


two_consecutive_hours(TL) :- member([Day,Hour1],TL),
  member([Day,Hour2],TL), H2 is Hour1+1, H2=Hour2.


three_classes_same_day(TL) :- member([Day,Hour1],TL),
  member([Day,Hour2],TL), member([Day,Hour3],TL),
  not(Hour1=Hour2), not(Hour1=Hour3), not(Hour2=Hour3).
```

```
member(X,[X|L]).
member(X,[Y|L]) :- member(X,L).
```

Here we use the backtracking version of the **member** predicate to both generate class times and to select class times from a list.

Computer vision often involves applying many constraints to interpret some picture. Preliminary visual processing (for example, the techniques of Section 9.16) groups parts of a picture into regions based on similarities of brightness, color, and texture. The remaining problem is to decide what each region of the picture represents. For this, variables correspond to regions, and values of the variables correspond to labels (from a finite label set) for the identity of each region. Additional constraints (conditions) affect which regions can border others, or be above others, or be inside others.

Consider the aerial photograph shown in Figure 13-2. Assume regions can be labeled either **grass**, **water**, **pavement**, **house**, or **vehicle**. Let's give some reasonable constraints about relationships of regions in an aerial photo.

> --a region cannot border or be inside a region with the same label (else we couldn't see the boundary line);
> --houses and vehicles cannot be next to or surrounded by water regions (we assume no water is deep enough for boats in this part of the world);
> --vehicles must be next to or surrounded by pavement, but pavement cannot be inside them;
> --vehicles cannot be inside houses;
> --grass cannot be inside vehicles or houses;
> --pavement cannot be completely inside another region (pavement nearly always connects to other pavement);
> --only houses, vehicles, and pavement are regular (straight-edged) regions;
> --only grass and water are irregular regions;
> --on a scale of 1 inch = 20 feet, vehicles cannot exceed an area of one quarter of a square inch.

These constraints might be violated, but aren't likely to be--remember, artificial intelligence programs don't try to be perfect, just "intelligent". We'll often know when they are violated, by being unable to find a solution. Then we can remove the most questionable constraint and try again, and then the second most questionable, and so on until we do find a solution.

Examining the photo in Figure 13-2, we see five regions; we've written names on them for convenience. R5 is the only region not "large" by the definition. Regions R3 and R5 are the only regions that are definitely regular; regions R1 and R2 are definitely irregular; but R3 is hard to classify so we won't state that it is either regular or irregular. R1 borders R2, and R2 borders R4; R3 is inside R2, and R5 is inside R4. So the photo interpretation problem can be described in a query this way:

```
?- label(R1), label(R2), label(R3), label(R4), label(R5),
  borders(R1,R2), borders(R2,R4), inside(R3,R2), inside(R5,R4),
  large(R1), large(R2), large(R3), large(R4),
  regular(R3), regular(R5), irregular(R2), irregular(R1).
```

with the following definitions:

```
label(grass).
label(water).
label(pavement).
label(house).
label(vehicle).


borders(A1,A2) :- not(A1=A2), not(water_constraint(A1,A2)),
  not(vehicle_constraint(A1,A2)).


inside(A1,A2) :- not(A1=A2), not(water_constraint(A1,A2)),
  not(vehicle_constraint2(A1,A2)),
  not(grass_constraint(A1,A2)), not(A1=pavement).

water_constraint(water,house).
water_constraint(water,vehicle).
water_constraint(house,water).
water_constraint(vehicle,water).


vehicle_constraint(A1,vehicle) :- not(A1=pavement).
vehicle_constraint(vehicle,A2) :- not(A2=pavement).
vehicle_constraint2(A1,vehicle).
vehicle_constraint2(vehicle,A2) :- not(A2=pavement).


grass_constraint(grass,house).
grass_constraint(grass,vehicle).


large(A) :- not(A=vehicle).


regular(house).
regular(vehicle).
regular(pavement).


irregular(grass).
irregular(water).
```

Here **water_constraint**, **vehicle_constraint**, and **grass_constraint** predicates specify permissible combinations of adjacent regions.

Rearranging long queries without local variables

Before processing a long query, we can often rearrange it into a more efficient form. It's usually best in an "and" to put first the predicate expressions hardest to satisfy. We can use probabilities to explain why. Take the query

```
?- a, b, c.
```

and suppose **a** is true with probability 0.9, **b** with probability 0.5, and **c** with probability 0.8. First, assume these probabilities are independent of each other. Then the preceding query has a probability of success of |0.9 * 0.5 * 0.8 = 0.36|, and a probability of failure of |0.64|. So it will more often fail than succeed. When it does fail, we would like to know as soon as possible. With the order as given, 0.1 of the time we will recognize a failure from evaluating just the **a**, and |0.9 * 0.5 = 0.45| from evaluating both **a** and **b**. That's a total of 0.55. But if we rearrange the query as

```
?- b, c, a.
```

then 0.5 of the time we will know a failure from examining the first predicate expression, and |0.5 * 0.2 = 0.1| from examining the first two. That's a total of 0.6. So we're finding the failure earlier, meaning a little less work for us.

In general: if we have a bunch of predicate expressions "and"ed together, we should order them by *increasing* probability of success. That way, if the whole thing is going to fail, we find out as soon as possible. This also applies to all rules without local variables. For instance:

```
f(X) :- b(X), c(X), a(X).
```

If for a random **X**, **b** succeeds with probability 0.5, **c** with probability 0.8, and **a** with probability 0.5, and these probabilities are all independent of each other, then this is the best arrangement of the right side of the rule for all queries of **f** with a bound argument, as for example

```
?- f(foo).
```

There is a corresponding idea for "or"s (rules "or"ed together). Consider:

```
r :- a.
r :- b.
r :- c.
```

Here we should try the rule most likely to succeed first, because if it succeeds then we're done. And we should try second the rule second-most-likely to succeed. The same applies to the rules

```
r(X) :- a(X).
r(X) :- b(X).
r(X) :- c(X).
```

provided **r** is only queried with **X** bound. So we should order "or"s by *decreasing* probability of success, just the opposite of "and"s.

Some mathematics

We can give general mathematical formulas for this rearrangement problem. They apply to any queries or rules without local variables, without "side effect" predicates like **asserta** and **consult**, and for which the success probabilities of expressions are independent.

First, suppose we have some expressions "and"ed together, where each has a probability of success |p sub i| and an execution cost |c sub i| (measured in arbitrary nonnegative units, perhaps time, perhaps the number of database queries made). Number the expressions from left to right. Consider two adjacent ones |i| and |i+1|. Interchanging them will improve computation cost if

```
c sub i   +   p sub i c sub i+1   >   c sub i+1   +   p sub i+1 c sub i
```

Manipulating the inequality we get:

```
( 1 - p sub i+1 ) c sub i   >   ( 1 - p sub i ) c sub i+1
```

or:

```
( 1 - p sub i ) / c sub i   <   ( 1 - p sub i+1 ) / c sub i+1
```

In other words, we can use the ratio of probability of failure to the execution cost as a sorting criterion for adjacent expressions. Since repeated interchanges of adjacent expresssions are sufficient to sort a list (this is how the famous "bubble sort" sorting algorithm works), we should sort the expressions by this criterion to get the best order in an "and".

A similar formula applies to "or"s. We should interchange adjacent expressions |i| and |i+1| in an "or" if

```
c sub i + ( 1 - p sub i ) c sub i+1   >   c sub i+1
   +   ( 1 - p sub i+1 ) c sub i
```

or:

```
p sub i / c sub i   <   p sub i+1 / c sub i+1
```

Here's an example. Take the query

```
?- a, b.
```

where **a** has probability 0.6 of success, and **b** has probability 0.8. If it costs the same amount of time to query either, **a** should be first. We can see this with the "and" formula because

```
( 1 - 0.6 ) / 1   >   ( 1 - 0.8 ) / 1   ,     "or"     0.4 > 0.2
```

But suppose (without changing its probability) that **a** is defined as

```
a :- c, d.
```

where **c** and **d** are only expressible as facts. Now it requires more work to find the truth of **a**: a query of **a**, followed by queries of **c** and **d**. If each query costs about the same, that's three times as much work. So now

```
(1 - 0.6 ) / 3   <   ( 1 - 0.8 ) / 1   ,     "since"     0.1333   "<"   0.2
```

and **b** is better to put first in the original query.

Rearranging queries with local variables

Local variables cause problems for the preceding reasoning. Consider:

```
?- a(X), b(X).
```

Here **X** is local to the query, and will be bound in **a**; in queries like these, **a** isn't expected to fail. But if we try to rearrange the query to

```
?- b(X), a(X).
```

Now **b** is the predicate that binds **X**, and it becomes unlikely to fail. So predicate expressions that bind local variables to values must be analyzed specially.

For certain expressions we can ignore this problem, those that require bound variable values. Consider:

```
?- a(X), b(Y), X>Y.
?- a(X), Y is X*X.
?- a(X), square(X,Y).
```

where **square** in the last query is defined by

```
square(X,Y) :- Y is X*X.
```

In each of these queries, putting last predicate expression first will cause an error message about trying to refer to an unbound variable. Things are almost as bad with

```
?- a(X), b(Y), not(c(X,Y)).
```

which, while not giving an error message, gives completely different answers when the last expression is put first (see a similar use in the **done** predicate for forward chaining in Chapter 7):

```
?- not(c(X,Y)), a(X), b(Y).
```

The first predicate expression here asks whether there are any two-argument **c** facts, and doesn't connect its **X** and **Y** to those in the **a** and **b** predicates.

But assuming we *can* move expressions to a front binding position, which one should we put there? Consider

```
?- a(X), b(X).
```

Suppose there are only a finite number of **X** values satisfying this query, say 5. Suppose predicate **a** is satisfied by 20 values of **X**, and predicate **b** by 100 values. If we put first **a**, then 5 of its 20 solutions will subsequently satisfy **b**. But if we put first **b**, then 5 of its 100 solutions will subsequently satisfy **a**. So if **X** values occur in a random way, we'll need more backtracking to answer the query the second way than the first. The amount of backtracking is the important criterion, because probabilities and costs don't apply here. Probabilities aren't relevant because we don't usually expect a variable-binding predicate to fail, and costs aren't relevant because binding a variable doesn't cost much.

To summarize: when you have several predicate expressions in an "and" that could bind a variable were they moved to the left in a query, move the one with the fewest possible bindings. Then sort the remaining expressions by increasing probability of success, as before.

Rearranging queries based on dependencies

Besides the preceding criteria, we should also use *dependencies* as criteria for rearranging queries. A dependency holds between two predicate expressions if they share a variable. Generally speaking, you should try to maximize the number of dependencies between adjacent expressions in an "and", while staying consistent with the previous criteria. This is because when two adjacent expressions don't have variables in common, the variable bindings in the first are irrelevant to the success or failure of the second, and backtracking may be wasting its time going to the second when a failure is due to the first.

For example, consider

```
?- a(X), b(Y), c(X), e(X,Y), d(X), f(Y), g(Y).
```

Here the **b** expression has no dependency on the **a** expression, the **c** none on the **b**, and the **f** none on the **d**, though the other adjacent expressions do have dependencies. A better rearrangement would be (assuming that this has about the same average overall cost due to expression probabilities and costs)

```
?- a(X), c(X), d(X), b(Y), e(X,Y), f(Y), g(Y).
```

where there is only one non-dependency between adjacent expressions, between **d** and **b**. Now when **g** fails, backtracking only needs to go back three expressions, whereas before it needed to go back five.

Summary of guidelines for optimal query arrangements

We can summarize what we have concluded about query and rule rearrangement by the following guidelines. (They're like heuristics because they can be wrong.)

1. For an "and" that does not bind local variables, sort the predicate expressions by decreasing values of the ratio of the failure probability for the expression to its execution cost.

2. For an "or" or for rules with the same left side (an implicit "or"), sort them by decreasing values of the ratio of the success probability for the rule to its execution cost.

3. For an "and" that binds local variables, choose to make binding for each variable the predicate expression that mentions it, among those that can bind it, both legally and without changing the predicate meaning, and that has the fewest database matches when queried alone. Then rearrange the rest of the

expressions by guidelines 1 and 3 while freezing the position of these binding expressions.

4. Maximize the number of dependencies between adjacent expressions in an "and", as long as you don't violate the binding choices made according to guideline 3, or significantly violate the probability-cost ratio ordering in guideline 1.

Let's apply these guidelines to the scheduling example of Section 13.1. It originally looked like this:

```
classtime(T1), classtime(T2), classtime(T3),
  classtime(T4), classtime(T5),
  not(occupied(T1)), not(occupied(T2)), not(occupied(T3)),
  not(occupied(T4)), not(occupied(T5)),
  not(two_consecutive_hours([T1,T2,T3,T4,T5]),
  not(three_classes_same_day([T1,T2,T3,T4,T5]).
```

For guideline 3 the question is which expressions should bind variables **T1**, **T2**, **T3**, **T4**, and **T5**. But every expression besides the **classtime** ones has a **not** in front of it, so this is the only arrangement that will satisfy the conditions of guideline 3.

To better satisfy guideline 4, we can move the **not** expressions to immediately after the expressions binding them.

```
classtime(T1), not(occupied(T1)),
classtime(T2), not(occupied(T2)),
classtime(T3), not(occupied(T3)),
classtime(T4), not(occupied(T4)),
classtime(T5), not(occupied(T5)),
not(two_consecutive_hours([T1,T2,T3,T4,T5]),
not(three_classes_same_day([T1,T2,T3,T4,T5]).
```

It's hard to guess success probabilities here--this varies enormously with the number of **occupied** facts we have--so we just use computation cost to order by guideline 1. So we should keep the last two expressions last because they both involve computation (calls to rule definitions), and the other **not** expressions don't. Similarly, **two_consecutive_hours** should precede **three_classes_same_day** because the first has three expressions in its definition, the second six.

Rearrangement and improvement of the photo interpretation query

The photo interpretation query of Section 13.1 requires more care to rearrange. Here it is:

```
?- label(R1), label(R2), label(R3), label(R4), label(R5),
  borders(R1,R2), borders(R2,R4), inside(R3,R2), inside(R5,R4)
  large(R1), large(R2), large(R3), large(R4),
  regular(R3), regular(R5), irregular(R2), irregular(R1).
```

All five variables (**R1**, **R2**, **R3**, **R4**, and **R5**) are bound locally in this query. So apply guideline 3 to decide which expressions to put first. The definitions of **borders**, **inside**, and **large** all have **not**s containing variables that would become unbound if those expressions were put first in the query. So the only expressions that can bind variables are those for **label**, **regular**, and **irregular**. Among them, **irregular** is best by guideline 3 because it has only two argument possibilities, **regular** is second best with three possibilities, and **label** is third with five (see Figure 13-3). That suggests this rearrangement:

```
?- irregular(R2), irregular(R1), regular(R3), regular(R5),
  label(R1), label(R2), label(R3), label(R4), label(R5),
  borders(R1,R2), borders(R2,R4), inside(R3,R2), inside(R5,R4)
  large(R1), large(R2), large(R3), large(R4).
```

But now **label** can never fail for **R2**, **R1**, **R3**, and **R5** (see Figure 13-3); any value bound in the first line will satisfy the **label** predicate. So we can eliminate four predicate expressions to give the equivalent shorter query:

```
?- irregular(R2), irregular(R1), regular(R3), regular(R5),
  label(R4), borders(R1,R2), borders(R2,R4), inside(R3,R2),
  inside(R5,R4) large(R1), large(R2), large(R3), large(R4).
```

(We must still keep **label(R4)** because there's no **regular** or **irregular** fact to bind **R4**.)

So now the first line does all the variable binding. Let's now apply guideline 1 (the probability-to-cost ratio ordering) to the rest of the query, as suggested in the discussion of guideline 3. To simplify analysis, we'll assume all labels are equally likely, and we'll measure processing cost by the number of queries needed. Predicate **large** (see Figure 13-3) succeeds except if its argument is **vehicle**, so it then succeeds with probability 0.8; it requires two query lookups. This means a guideline-1 ratio of |(1 - 0.8) / 2 = 0.1|. Predicates **borders** and **inside** require more complicated analysis, however. Here again are their definitions:

```
borders(A1,A2) :- not(A1=A2), not(water_constraint(A1,A2)),
  not(vehicle_constraint(A1,A2)).
inside(A1,A2) :- not(A1=A2), not(water_constraint(A1,A2)),
  not(vehicle_constraint2(A1,A2)), not(grass_constraint(A1,A2)),
  not(A1=pavement).
```

Counting both left and right sides of rules, **borders** requires four query lookups for its four expressions, and **inside** requires six. (We'll ignore the added complication of lowered cost due to incomplete rule processing for some rule failures.) From Figure 13-4, we can estimate the probability of success of **borders** as |12 / 25 = 0.48|, and the probability of success of **inside** as |7 / 25 = 0.28|. Then the ratios needed for guideline 1 are |(1 - 0.48) / 4 = 0.13| for **borders** and |(1 - 0.28) / 6 = 0.18| for **inside**. By these rough numbers, **inside** expressions should come first, then **borders** expressions, and then **large** expressions. So the query should be rearranged to:

```
?- irregular(R2), irregular(R1), regular(R3), regular(R5),
  label(R4), inside(R3,R2), inside(R5,R4), borders(R1,R2),
  borders(R2,R4), large(R1), large(R2), large(R3), large(R4).
```

Studying Figure 13-3, we see that **large** succeeds whenever **irregular** succeeds. So we can eliminate the **large(R1)** and **large(R2)** to get an equivalent query:

```
?- irregular(R2), irregular(R1), regular(R3), regular(R5),
  label(R4), inside(R3,R2), inside(R5,R4), borders(R1,R2),
  borders(R2,R4), large(R3), large(R4).
```

Now we can apply guideline 4, trying to group expressions mentioning the same variable together. One way is to move the **inside**, **borders**, and **large** expressions back into the first line, preserving their order at the expense of the order of the first line (since guideline 1 order is more important than guideline 3 order):

```
?- irregular(R2), regular(R3), inside(R3,R2),
  regular(R5), label(R4), inside(R5,R4),
  irregular(R1), borders(R1,R2), borders(R2,R4),
  large(R4), large(R3).
```

Here we've had to move **irregular(R1)** considerably to the right, but that's the only expression moved significantly right.

So now we've probably got a much faster query. We emphasize "probably", since we had to make a number of simplifying assumptions about probabilities and costs. (If these assumptions seemed too much, we can always do a more careful analysis using conditional probabilities and more subcases.)

Dependency-based backtracking

Our second technique for handling long queries is *dependency-based backtracking*. Consider the query:

```
?- a(X), b(X,Y), c(Z), d(X,Z), e(X).
```

Suppose the probability of any predicate expression succeeding is 0.1, and that these probabilities are independent of the success or failure of other expressions. On the average we'll need to generate 10 **X** values before we find one that works, 10 **Y** values, and 10 **Z** values. But for each **X** value we must try about 10 **Y** values, for about 100 **Y** values in all; and for each **Y** value we must try about 10 **Z** values, for about 1000 about **Z** values in all. So there will be a lot of backtracking to answer this query with Prolog's standard backward-chaining control structure.

An alternative is *nonchronological* or *dependency-based* backtracking. That is, backing up to the expression that did the last variable binding that could have affected the failure, and finding a new binding there. That's not necessarily the previous expression, so dependency-based backtracking is "smarter" backtracking, backtracking that notices and takes obvious shortcuts. Dependencies (indications of which expressions have common variables) help this considerably, so that's why they're in the name of the technique.

Let's apply dependency-based backtracking to the previous query. First, we need to tabulate the predicate expressions in the query that bind, and what expressions have dependencies on what other expressions; see Figure 13-5. Assume as a database for this query:

```
a(1).
a(2).
a(3).
b(A,B) :- B is 3*A.
c(4).
c(1).
d(A,B) :- A>B.
e(3).
```

Then what happens in executing the query with dependency-based backtracking is shown in Figure 13-6. Notice that it's identical to regular Prolog backtracking except at three places, after steps 7, 14, and 16:

> --After step 7, we backtrack from the third predicate expression directly to the first, since variable **X** must be responsible for the earlier failure of the fourth expression, and variable **Y** has nothing to do with that failure.
>
> --After step 14, we backtrack from the last expression directly to the first, since variable **X** is the only possible cause of the failure, and the first expression binds it.
>
> --After step 16, we skip over the third expression in moving back to the right, since it didn't fail and it doesn't include any variables whose values were changed since the last time it was visited.

So the effect of dependency-based backtracking is to save us 6 actions (right and left movements) out of 24 total. That's not much, but other queries show more dramatic savings.

Here's an algorithm for dependency-based backtracking:

```
1. Mark every predicate expression in the query as "active". Set P to the leftmost predicate in the query.

2. Execute repeatedly until you reach the right end of the query, or backtrack off the left end:

    (a) Execute expression P, and mark it as "inactive".

    (b) If P succeeds and does not bind variables, set P to the next expression to the right that
    is marked "active".

    (c) If P succeeds and binds variables, mark as "active" all other query expressions to the
    right containing those variables if (1) they are not so marked already and (2) the values are
    different from the previous values of those variables. Set P to the next expression to the
    right that is marked "active".

    (d) If P fails, mark as "active" all the expressions to the left of P that bind variables
    mentioned in P. Set P to the first expression to the left that is currently marked "active"
    (which is not necessarily one just marked, for it may have been marked previously).
```

There are further improvements on dependency-based backtracking that we don't have space for here. The theory is quite elegant, and this is an active area of research, also called *truth maintenance*.

Reasoning about possibilities

There's a third way to efficiently handle long queries. The idea is to abandon considering variable bindings one at a time, and to reason about possibilities for each variable. Consider the query of the last section:

```
?- a(X), b(X,Y), c(Z), d(X,Z), e(X).
```

Suppose there are only a finite number of ways to satisfy each predicate expression. We could figure out all the possible values for **X** that satisfy **a(X)**, and then check which of them also satisfy e(X). Then for each remaining choice we could check which could satisfy **b(X,Y)**, crossing out ones that can't. Then we could figure out the possible values of **Z** that satisfy **c(Z)**, and which of those **Z** values for which there is a corresponding **X** value remaining that can satisfy **d(X,Z)**. Sherlock Holmes might put the approach this way: when you have eliminated the impossible, whatever remains, however improbable, must include the truth.

Why reason about possibilities for variables? So we don't need to work as hard to answer the query. Early elimination of impossible values for variables reduces fruitless backtracking. Sometimes we can even reduce possibilities to one for each variable, and we don't need to do any further work. The idea is called *relaxation*--a technical term that has nothing to do with vacations on beaches. Artificial intelligence uses in particular *discrete relaxation*, relaxation involving variables with finitely-long possibility lists.

Using relaxation for the photo interpretation example

We'll present an algorithm for relaxation in the next section. It's a little complicated. To work our way up to it, let's see how we can reason about possibility lists for the photo interpretation problem of Section 13.1, the one with the query:

```
?- label(R1), label(R2), label(R3), label(R4), label(R5),
  borders(R1,R2), borders(R2,R4), inside(R3,R2), inside(R5,R4),
  large(R1), large(R2), large(R3), large(R4), regular(R3),
  regular(R5), irregular(R2), irregular(R1).
```

To get possibility lists for each variable, we must extract the single-variable expressions referring to each variable:

```
variable R1: label(R1), large(R1), irregular(R1).
variable R2: label(R2), large(R2), irregular(R2).
variable R3: label(R3), large(R3), regular(R3).
variable R4: label(R4), large(R4).
variable R5: label(R5), regular(R5).
```

Reexamining Figure 13-3, we can see what values (labels) are consistent with the expressions. The possible labels are grass, water, pavement, house, and vehicle. The **large** predicate rules out the vehicle label. The **regular** predicate specifies house, vehicle, or pavement, and **irregular** specifies grass or water. So the initial possibility lists are:

```
variable R1: grass or water
variable R2: grass or water
variable R3: house or pavement
variable R4: house or pavement or grass or water
variable R5: house or vehicle or pavement
```

Now let's eliminate or "cross out" possibilities using the remaining multivariable predicate expressions (constraints) of the original query:

```
borders(R1,R2), borders(R2,R4), inside(R3,R2), inside(R5,R4)
```

We can pick a variable and a possibility, and check if the possibility works. Suppose we pick **R1** and **grass**. The **borders(R1,R2)** mentions **R1**, so we see if some **R2** value will work with **R1=grass**--that is, whether there's some label that could border grass. Yes, **R2=water** works. Check the definition of **borders**:

```
borders(A1,A2) :- not(A1=A2), not(water_constraint(A1,A2)),
  not(vehicle_constraint(A1,A2)).
```

For **A1=grass** and **A2=water**, **A1** is not equal to **A2**. The **water_constraint** only applies when one label is a house or vehicle, as you can see from its definition:

```
water_constraint(water,house).
water_constraint(water,vehicle).
water_constraint(house,water).
water_constraint(vehicle,water).
```

And **vehicle_constraint** only applies when **A1** or **A2** is a vehicle:

```
vehicle_constraint(A1,vehicle) :- not(A1=pavement).
vehicle_constraint(vehicle,A2) :- not(A2=pavement).
```

So **R1=grass** will work for at least one value of **R2**, which is all we need to keep us from crossing it out.

But if we examine a different variable and value, things can be different. Suppose we pick the **R2** variable and **water**. The **inside(R3,R2)** constraint mentions **R2**. But there's no way to label **R3** to satisfy the second expression on the right side of the **inside** definition. Here's that definition:

```
inside(A1,A2) :- not(A1=A2), not(water_constraint(A1,A2)),
  not(vehicle_constraint2(A1,A2)), not(grass_constraint(A1,A2)),
  not(A1=pavement).
```

The only possibilities for **R3** from the single-variable constraints are house and vehicle, and water_constraint succeeds for both. Hence the **not** fails, and **inside** fails too because this is the only rule for **inside**. In other words, a vehicle or house can't be inside water, a reasonable assumption for most pictures, though you can probably imagine a few exceptions. So **R2** cannot be water. The only remaining possibility is grass, so **R2** must be grass.

Once we've eliminated possibilities for a variable, we can often eliminate possibilities for a variable appearing in constraints with it. Consider variable **R1** which we didn't have any success with originally. Now it's no longer possible that **R1** is grass, because then **R1** would be the same as **R2**, and the first condition in the **borders** definition prohibits that. It says two adjacent regions can't have the same label since we couldn't then see the boundary. There's only one remaining label for **R1**, **water**, so **R1** must be that.

Our possibility lists are now:

```
variable R1: water
variable R2: grass
variable R3: house or pavement
variable R4: house or pavement or grass or water
variable R5: house or vehicle or pavement
```

Now consider **R3**. It can't be pavement because **R3** is inside **R2**, and the last condition in the definition of **inside** says that the thing inside can't be pavement. That is, pavement must connect to other pavement--a reasonable, though sometimes false, assumption. So **R3** must be a house.

Consider **R4**. It can't be grass because it borders **R2**, and **R2** is already grass. It can't be water because water can't contain (as **R5**) a house or a vehicle by **water_constraint**, and water can't contain pavement by the last condition in the **inside** definition. That leaves house or pavement for **R4**.

Now consider **R5**. It can't be pavement because pavement can't be contained in anything. So it must be a house or vehicle.

Now consider **R4** again. Suppose it's a house. Then **R5** can't be a house too because that would make two adjacent regions with the same label. But **R5** couldn't be a vehicle either, because a vehicle can't be inside a house by the **vehicle_constraint2** definition. So **R4** can't be a house, and must be pavement.

No further eliminations can be done, so the final possibility lists are:

```
variable R1: water
variable R2: grass
variable R3: house
variable R4: pavement
variable R5: house or vehicle
```

Quantifying the effect (*)

To show the savings in considering possibility lists instead of individual values, let's use some numbers. Suppose there are |n sub Y| possibilities for a variable **Y** . Suppose the expression **d(X,Y)** is satisfied by fraction p of all possible **XY** pairs. If we can assume satisfaction pairs for d are randomly distributed over all **XY** pairs, then the probability that some **X** value r has at least one corresponding **Y** that satisfies d can be approximated by the binomial series:

```
1 - ( 1 - p ) sup { n sub Y }  =    { n sub Y } p   -
{ n sub Y } ( { n sub Y }  - 1 ) p sup 2 / 2   +
{ n sub Y } ( { n sub Y } - 1 ) ( { n sub Y }  - 2 ) p sup 3 / 6   -   . . .
```

And the expected number of such matched **X** values in the |n sub X| values for **X** will be just |n sub X| times this. Figure 13-7 shows the probability computed for sample values of p and n. When p is small (as a rule of thumb, whenever p < 0.2) we can approximate the binomial distribution by the Poisson distribution, and the probability is

```
1 - e sup { - n sub Y p }
```

And the expected number of **X** values remaining from |n sub X| originally, after examining predicate **d(X,Y)** which has probability of success p and |n sub Y| values of **Y** that can match it is

```
n sub X ( 1 - e sup { - n sub Y p } )
```

The smaller this number, the better is reasoning about possibilities first before working Prolog-style.

If several constraints mention a variable, each can independently eliminate possibilities, further reducing their number. Consider this query:

```
?- a(R), b(S), c(T), d(R,S), e(S,T).
```

Suppose there are 10 ways to satisfy predicate **a**, 20 ways to satisfy **b**, and 50 ways to satisfy **c**. Suppose further that the probability is 0.1 that a random pair of values satisfy **d**, and 0.01 that a random pair of values will satisfy **e**, and that probabilities are all independent. Then:

> --From Figure 13-7 and our formula we can expect |10 * 0.88 = 8.8| possibilities for **R** that satisfy **d**, on the average.
>
> --And we can expect |50 * 0.39 = 19.5| possibilities for **T** that satisfy **e**, on the average.
>
> --**S** occurs in both the two-argument predicates **d** and **e**. For the occurrence in **d** we can expect |20 * (1 - e sup {-8.8 * 0.1}) = 11.7| possibilities remaining.
>
> --The predicate **e** can only remove further **S** possibilities of these. If satisfaction of **e** is probabilistically independent of satisfaction of **d**, for instance, we expect |11.7 * (1 - e sup {-19.5 * 0.01}) = 2.1| possibilities remaining for **S**. That's a lot better than the initial 20.

Formalization of pure relaxation

Here's an algorithm for the simplest kind of relaxation method for answering many-constraint queries: *pure* relaxation. Note for efficiency in this algorithm that it's important to index all variable names appearing in constraint expressions.

> 1. Create possibility lists for each variable using all single-variable query predicate expressions. (Single-argument expressions with a variable argument are single-variable, but others like **p(X,a,X)** can be too.) In Prolog, **bagof** will help do this.
>
> 2. Mark each variable in the original query as "active".
>
> 3. Do the following step repeatedly until no more "active" variables remain.
>
> > (a) Choose some "active" variable A. (Heuristics may help choose, like "focus-of-attention" heuristics.) For each current possible value V for A:
> >
> > > For each constraint C mentioning variable A:
> > >
> > > > Check to see if when variable A has value V, constraint C can be satisfied some way (binding other variables of C as necessary). If not, remove V from the possibility list for A; otherwise do nothing. Throw away previous bindings for other constraints when considering each new constraint (this makes the relaxation "pure").
> >
> > (b) Mark variable A as "inactive", and mark as "active" all variables (i) mentioned in

constraints with A, and (ii) marked "inactive", and (iii) having more than one possibility remaining. Eliminate any constraints for which all the variables have unique values now.

4. If there is now a unique possibility for each variable, stop and give those values as the query answer. Otherwise, run the Prolog interpreter on the original query, drawing values from the possibility lists.

We emphasize that for every constraint we consider, we ignore bindings of its variables made when those variables occurred in other constraints. This "pure" relaxation idea may seem wasteful, but if we didn't do it we would just be doing Prolog-style backward chaining in disguise. But compromises between the two approaches are possible, like the *double relaxation* we'll show later.

One nice thing about this relaxation algorithm is its good potential for concurrency. We can work in parallel on different constraints, different values, or even different variables.

Another relaxation example: cryptarithmetic

No artificial intelligence book would be complete without an analysis of a good puzzle problem. Analysis of "brain-teaser" puzzles stimulated much of the early development of the field.

A classic application of relaxation is to puzzles in which numbers are represented by letters (*cryptarithmetic*). Here the variables are the letters and the possibility lists are numbers those letters can be (often, the possibilities are restricted to single digits). Constraints come from arithmetic relationships between the digits. Here's a famous example, where every letter stands for a different digit, and every occurrence of a letter is the same digit.

```
    S E N D
+ M O R E
______________________
M O N E Y
```

Note that's the letter "O", not the number zero. To make the problem easier, suppose we are told that E=5. (Otherwise we can't solve it completely with pure relaxation.)

So from the single-variable constraints, initial possibility lists for this problem are:

```
M: [1,2,3,4,5,6,7,8,9]
S: [1,2,3,4,5,6,7,8,9]
O: [0,1,2,3,4,5,6,7,8,9]
E: [5]
N: [0,1,2,3,4,5,6,7,8,9]
R: [0,1,2,3,4,5,6,7,8,9]
D: [0,1,2,3,4,5,6,7,8,9]
Y: [0,1,2,3,4,5,6,7,8,9]
```

Each letter represents a variable for this problem. Note M and S cannot be 0 because they are the first digits in numbers, and numbers don't start with 0. On the other hand, we can't cross out the 5's yet because we need to use a multi-variable constraint for them.

But there are also some important additional variables buried in the problem: the carries from each column. We'll label these C1, C10, and C100, representing the carries out of the 1's column, the 10's column, and the 100's column respectively. Their possibilities for two-number addition are:

```
C1: [0,1]
C10: [0,1]
C100: [0,1]
```

Here are the constraints on the variables. The first four come from the rules of column addition, and the last says that digit assignments must be unique. While these constraints could also be satisfied for integers greater than 10 or real numbers, the initial possibility lists will prevent such happenings.

> Con1: D + E = Y + (10 * C1)
> Con2: N + R + C1 = E + (10 * C10)
> Con3: E + O + C10 = N + (10 * C100)
> Con4: S + M + C100 = O + (10 * M)
> Con5: all_different_numbers(S,E,N,D,M,O,R,Y)

Since E=5 we can substitute that value into the constraints to simplify them:

> Con1: D + 5 = Y + (10 * C1)
> Con2: N + R + C1 = 5 + (10 * C10)
> Con3: 5 + O + C10 = N + (10 * C100)
> Con4: S + M + C100 = O + (10 * M)
> Con5: all_different_numbers(S,5,N,D,M,O,R,Y)

That covers step 1 of the relaxation algorithm, so we proceed to step 2 and mark every letter and carry as "active". Now we start step 3, the big step. We will follow a "leftmost-first" heuristic that says to choose to work on the active variable representing the digit or carry farthest left in the sum display.

1. We can pick either M or C1000 as the first variable according to our heuristic; suppose we pick M. Only constraints Con4 and Con5 mention M. And the only value of M satisfying Con4 is 1, since 0 is not on M's original possibility list, and a value for M of 2 or greater would make the right side of Con4 at least 20, and there's no combination of S and C100 drawn from their possibility lists whose sum plus 2 (for M=2) could equal or surpass 20. We mark M inactive.

2. Using our "leftmost active" choice heuristic, we pick S next. It's only mentioned in Con4 and Con5. The only way to satisfy Con4 is if S is 8 or 9, because M is 1 and the right side is at least 10. Con5 doesn't further eliminate possibilities for S. We mark S inactive.

3. We pick O next. It's mentioned in Con3, Con4, and Con5. We can't do anything with Con3, but by Con4 we see O must be 0 or 1 since M=1 and S is either 8 or 9. But turning now to Con5, O cannot be 1 (with M already 1). So O=0. We mark S active and O inactive.

4. We pick S next. We can't conclude anything about it, so we mark it inactive. (Nothing is marked active because we didn't rule out any possibilities.)

5. We pick C100 next. It's mentioned in Con3 and Con4. It must be 0, since |5 + 0 + C10 < 7|. We mark S

active, and C100 inactive.

6. We pick S next. Now by Con4 it must be 9, since |9 + 1 + 0 = 0 + (10 * 1)|. S is marked inactive.

Current possibility lists and activity markings are:

```
M: [1] (inactive)
S: [9] (inactive)
O: [0] (inactive)
E: [5] (inactive)
N: [0,1,2,3,4,5,6,7,8,9] (active)
R: [0,1,2,3,4,5,6,7,8,9] (active)
D: [0,1,2,3,4,5,6,7,8,9] (active)
Y: [0,1,2,3,4,5,6,7,8,9] (active)
C1: [0,1] (active)
C10: [0,1] (active)
C100: [0] (inactive)
```

And the sum looks like:

```
    9 5 N D
+ 1 0 R 5
______________
1 0 N 5 Y
```

And the constraints are:

Con1: D + 5 = Y + (10 * C1)
Con2: N + R + C1 = 5 + (10 * C10)
Con3: 5 + C10 = N
Con4: 10 = 10
Con5: all_different_numbers(9,5,N,D,1,0,R,Y)

We can just ignore Con4 from now on because it no longer has variables.

7. Pick N next. It's mentioned in Con2, Con3, and Con5. Con2 doesn't help, but Con3 says N must be 5 or 6. And then Con5 says N can't be 5 since E is already, so N=6. Mark N inactive.

8. Pick C10. It's mentioned in Con2 and Con3. But for Con3 it must be 1 since N=6. Mark C10 inactive.

9. Pick R. It's mentioned in Con2 and Con5. In Con2, |6 + R + C1 = 5 + (10 * 1)|, so R is 8 or 9. But by Con5, R cannot be 9 because S is already 9. So R=8. Mark R inactive.

10. Pick C1. It's mentioned in Con1 and Con2. By Con2, it must be 1. Mark C1 inactive.

11. Pick D. It's mentioned in Con1 and Con5. By Con1, |D + 5 = Y + (10 * 1)|. This means |D = Y +

5|, so D can only be 5, 6, 7, 8, or 9. But by Con5, the values 5, 6, 8, and 9 are ruled out because variables already have those values. So D=7. Mark D inactive.

12. Pick Y. By Con1, it must be 2. Mark Y inactive. Now every variable is marked inactive, so we can stop.

The final solution is:

```
M=1, S=9, O=0, E=5, N=6, R=8, D=7, Y=2, C1=1, C10=1, C100=0
```

```
    9 5 6 7
+ 1 0 8 5
__________________
 1 0 6 5 2
```

Notice relaxation is much like iterative methods for solving mathematical equations: "convergence" or progress towards the solution may be slow at times, but things speed up as the solution approaches.

Implementation of pure relaxation (*)

We can implement relaxation as we implemented the chaining programs (Chapter 7) and search programs (Chapter 10): with a problem-independent file and a problem-dependent file. The last part must specify the constraints. In it we'll use the following definitions (as either facts or rules):

choices(<variable_name>,<possible_value>)--states initial possibilities for <variable_name>, incorporating all the single-variable constraints

constraint(<type>,<argument_list>)--states multivariable constraints, classifying by type and arguments

satisfiable(<type>,<argument_list>,<bindings_found>)--determines whether a multi-variable constraint of a given type is actually satisfiable by some set of variable bindings

To circumvent the Prolog interpreter's usual binding of variables, "variable names" here must actually be constants (written in lower case). Note that because constraints have a type argument, all constraints of the same type can be checked by a single satisfiable predicate definition. Another reason for having both constraint and satisfiable is that relaxation can notice common variables between constraints without having to test constraints.

Here's an example problem-dependent file, the definitions for the photo interpretation problem considered earlier. Much of the code remains the same as before.

```
choices(R,CL) :- member(R,[r1,r2,r3,r4,r5]),
  bagof(C,choice(R,C),CL).


choice(R,grass) :- irregular(R).
choice(R,water) :- irregular(R).
choice(R,pavement) :- regular(R).
choice(R,house) :- regular(R).
choice(R,vehicle) :- regular(R), not(large(R)).
choice(R,L) :- not(regular(R)), not(irregular(R)), label(L).


label(grass).
label(water).
label(pavement).
label(house).
label(vehicle).


large(r1).
large(r2).
large(r3).
large(r4).


regular(r3).
regular(r5).


irregular(r2).
irregular(r1).

constraint(borders,[r1,r2]).
constraint(borders,[r2,r4]).
constraint(inside,[r3,r2]).
constraint(inside,[r5,r4]).


satisfiable(borders,Args,[A1,A2]) :- some_bindings(Args,[A1,A2]),
  not(A1=A2), not(water_constraint(A1,A2)),
  not(vehicle_constraint(A1,A2)).
satisfiable(inside,Args,[A1,A2]) :- some_bindings(Args,[A1,A2]),
  not(A1=A2), not(water_constraint(A1,A2)),
  not(vehicle_constraint2(A1,A2)),
  not(grass_constraint(A1,A2)), not(A1=pavement).


water_constraint(water,house).
water_constraint(water,vehicle).
water_constraint(house,water).
water_constraint(vehicle,water).


vehicle_constraint(A1,vehicle) :- not(A1=pavement).
vehicle_constraint(vehicle,A2) :- not(A2=pavement).
vehicle_constraint2(A1,vehicle).
vehicle_constraint2(vehicle,A2) :- not(A2=pavement).


grass_constraint(grass,house).
grass_constraint(grass,vehicle).
```

The problem-independent code translates the algorithm given in Section 13.12. Figure 13-8 gives the predicate hierarchy. The top-level predicate is relax of no arguments. It firsts converts all choices definitions into possibility lists. It then repeatedly chooses an active variable, until no more such variables exist. For that variable, it retrieves the possibility list and checks each possibility in turn, making a list of those possibilities that are satisfiable, using the bagof predicate defined in Section 10.6. If any previous possibilities are now impossible (meaning that the possibility list is shortened), it marks all variables in constraints with this one as active, and makes this variable inactive. If only one possibility remains for the variable, then (predicate check_unique) that value is substituted into the constraints. Any constraints no longer having variables in them, just constants, are removed.

Checking of possibilities is done by possible_value and constraint_violation. The first generates possibilities, and the second substitutes them into the constraints and calls satisfiable to check for satisfiability. Only constraints containing the variable under study are substituted into. Note it's important we check constraint violation and not constraint satisfaction; as soon as we find a constraint that can't be satisfied, we can stop and cross out a possibility.

Here's the program for pure relaxation:

```
/* Top-level routines */
relax :- setup, relax2.
relax :- listing(choice_list).


relax2 :- repeatactive, choice_list(O,CL), active(O),
  retract(active(O)), not(one_possibility(CL)),
  write('Studying variable '), write(O), nl,
  bagof(V,possible_value(O,CL,V),NCL), not(CL=NCL),
  retract(choice_list(O,CL)), asserta(choice_list(O,NCL)),
  write('Possibilities for '), write(O), write(' reduced to '),
  write(NCL), nl, update_activity(O), check_unique(O,NCL), fail.


/* Creation of the initial possibility lists */
setup :- abolish(choice_list,2), fail.
setup :- choices(O,VL), assertz(choice_list(O,VL)),
  assertz(active(O)), fail.
setup.


/* Analysis of a particular variable value */
possible_value(O,CL,V) :- member(V,CL),
  not(constraint_violation(V,O)).


constraint_violation(V,O) :- constraint(Type,Args), member(O,Args),
  substitute(O,V,Args,Args2), not(satisfiable(Type,Args2,Newargs)).


update_activity(O) :- constraint(T,A), member(O,A),
  member(O2,A), not(O2=O), choice_list(O2,L), not(active(O2)),
  not(one_possibility(L)), asserta(active(O2)), fail.
update_activity(O).


check_unique(O,[V]) :- constraint(Type,Args), member(O,Args),
  substitute(O,V,Args,Args2), retract(constraint(Type,Args)),
  some_var(Args2), asserta(constraint(Type,Args2)), fail.
check_unique(O,L).

/* Utility routines */
some_var([X|L]) :- choice_list(X,CL), !.
some_var([X|L]) :- some_var(L).


one_possibility([ZZZ]).


repeatactive.
repeatactive :- active(X), repeatactive.


member(X,[X|L]).
member(X,[Y|L]) :- member(X,L).


substitute(X,Y,[],[]).
substitute(X,Y,[X|L],[Y|L2]) :- substitute(X,Y,L,L2), !.
substitute(X,Y,[Z|L],[Z|L2]) :- substitute(X,Y,L,L2).


/* Note--the following is not necessary to run the program, but */
/* is provided as an aid to user definition of "satisfiable" */
some_bindings([],[]).
some_bindings([Arg|Args],[Substarg|Substargs]) :-
  choice_list(Arg,CL), !, member(Substarg,CL),
  substitute(Arg,Substarg,Args,Args2),
  some_bindings(Args2,Substargs).
some_bindings([Arg|Args],[Arg|Substargs]) :-
  some_bindings(Args,Substargs).
```

Here's the program working on our photo interpretation problem:

```
?- relax.
Studying variable r1
Studying variable r2
Possibilities for r2 reduced to [grass]
Studying variable r3
Possibilities for r3 reduced to [house]
Studying variable r4
Possibilities for r4 reduced to [pavement]
Studying variable r5
Possibilities for r5 reduced to [house,vehicle]
Studying variable r1
Possibilities for r1 reduced to [water]

choice_list(r1,[water]).
choice_list(r5,[house,vehicle]).
choice_list(r4,[pavement]).
choice_list(r3,[house]).
choice_list(r2,[grass]).

yes
```

Running a cryptarithmetic relaxation (*)

Here's a similar problem-dependent file defining the "send more money" cryptarithmetic problem:

```
choices(D,[0,1,2,3,4,5,6,7,8,9]) :- member(D,[n,d,r,o,y]).
choices(D,[1,2,3,4,5,6,7,8,9]) :- member(D,[s,m]).
choices(e,[5]).
choices(C,[0,1]) :- member(C,[c1,c10,c100]).


constraint(sum,[s,m,c100,o,m]).
constraint(sum,[e,o,c10,n,c100]).
constraint(sum,[n,r,c1,e,c10]).
constraint(sum,[d,e,0,y,c1]).
constraint(unique,[s,e,n,d,m,o,r,y]).


satisfiable(sum,L,[D1,D2,Carryin,Sum,Carryout]) :-
  some_bindings(L,[D1,D2,Carryin,Sum,Carryout]),
  S is D1 + D2 + Carryin,
  S2 is Sum + (Carryout * 10), S=S2.
satisfiable(unique,L,VL) :- not(duplication(L)),
  unique_values(L,VL,[]).


duplication([X|L]) :-  member(X,L), !.
duplication([X|L]) :- duplication(L).


unique_values([],[],KL).
unique_values([O|L],[V|L2],KL) :- choice_list(O,VL), !,
  unique_values(L,L2,KL), member(V,VL), not(member(V,L2)),
  not(member(V,KL)).
unique_values([K|L],[K|L2],KL) :- unique_values(L,L2,[K|KL]),
  not(member(K,L2)).
```

The satisfiable rule with sum as first argument checks column-sum constraints, and the satisfiable rule with unique checks unique-assignment constraints.

Here's a run. We've inserted extra blank lines for readability. Notice that the order of considering variables is different than for the problem analysis of Section 13.13, so possibility eliminations come in a different order.

```
?- relax.
Studying variable n
Possibilities for n reduced to [0,1,2,3,4,6,7,8,9]
Studying variable d
Possibilities for d reduced to [0,1,2,3,4,6,7,8,9]
Studying variable r
Possibilities for r reduced to [0,1,2,3,4,6,7,8,9]
Studying variable o
Possibilities for o reduced to [0,1]
Studying variable y
Possibilities for y reduced to [1,2,3,4,6,7,8,9]
Studying variable s
Possibilities for s reduced to [8,9]
Studying variable m
Possibilities for m reduced to [1]
Studying variable c1
Studying variable c10
Studying variable c100
Possibilities for c100 reduced to [0]

Studying variable s
Possibilities for s reduced to [9]
Studying variable y
Possibilities for y reduced to [2,3,4,6,7,8]
Studying variable o
Possibilities for o reduced to [0]
Studying variable r
Possibilities for r reduced to [2,3,4,6,7,8]
Studying variable d
Possibilities for d reduced to [2,3,7,8]
Studying variable n
Possibilities for n reduced to [6]
```

```
Studying variable c1
Possibilities for c1 reduced to [1]
Studying variable c10
Possibilities for c10 reduced to [1]

Studying variable d
Possibilities for d reduced to [7,8]
Studying variable r
Possibilities for r reduced to [8]
Studying variable y
Possibilities for y reduced to [2,3]
Studying variable d
Possibilities for d reduced to [7]
Studying variable y
Possibilities for y reduced to [2]

choice_list(y,[2]).
choice_list(d,[7]).
choice_list(r,[8]).
choice_list(c10,[1]).
choice_list(c1,[1]).
choice_list(n,[6]).
choice_list(o,[0]).
choice_list(s,[9]).
choice_list(c100,[0]).
choice_list(m,[1]).
choice_list(e,[5]).
```

Implementing double relaxation (*)

But pure relaxation can't solve the "send more money" problem unless we're told E=5 in advance. A better relaxation program could consider pairs of constraints together when it stops making progress analyzing single constraints. That is, a program could use the same bindings for variables common to two constraints. This "double relaxation" will be slower than pure relaxation, since there are more pairs than singles and it's doing more work with each pair, so it should only be used as a last resort. Only when both methods fail should relaxation stop.

```
/* Top-level routines */
relax :- setup, relax2.


relax2 :- single_pred_relax.
relax2 :- done.
relax2 :- double_pred_relax.
relax2 :- active(X), relax2.
relax2 :- not(active(X)), listing(choice_list).
```

```
done :- not(possibilities_remain), listing(choice_list).


possibilities_remain :- choice_list(O,CL),
  not(one_possibility(CL)).


single_pred_relax :- write('Single predicate relaxation begun.'),
  nl, repeatactive, choice_list(O,CL), active(O),
  retract(active(O)), not(one_possibility(CL)),
  write('Studying variable '), write(O), nl,
  bagof(V,possible_value(O,CL,V),NCL), not(CL=NCL),
  new_possibilities(O,CL,NCL), fail.


double_pred_relax :- write('Double predicate relaxation begun.'),
  nl, choice_list(O,CL), not(one_possibility(CL)),
  write('Studying variable '), write(O), nl,
  bagof(V,possible_value2(O,CL,V), NCL),
  not(CL=NCL), new_possibilities(O,CL,NCL), fail.


new_possibilities(O,CL,NCL) :- retract(choice_list(O,CL)),
  asserta(choice_list(O,NCL)), write('Possibilities for '),
  write(O), write(' reduced to '), write(NCL), nl,
  update_activity(O), check_unique(O,NCL).


/* Creation of the initial possibility lists */
setup :- abolish(choice_list,2), fail.
setup :- choices(O,VL), assertz(choice_list(O,VL)),
  assertz(active(O)), fail.
setup.


/* Single-relaxation analysis of a particular value for a particular variable */
possible_value(O,CL,V) :- member(V,CL),
  not(constraint_violation(V,O)).


constraint_violation(V,O) :- constraint(Type,Args),
  member(O,Args), substitute(O,V,Args,Args2),
  not(satisfiable(Type,Args2,Newargs)).


update_activity(O) :- constraint(T,A), member(O,A), member(O2,A),
  not(O2=O), choice_list(O2,L), not(active(O2)),
  not(one_possibility(L)), asserta(active(O2)), fail.
update_activity(O).
```

```
check_unique(O,[V]) :- constraint(Type,Args), member(O,Args),
  substitute(O,V,Args,Args2), retract(constraint(Type,Args)),
  some_var(Args2), asserta(constraint(Type,Args2)), fail.
check_unique(O,L).


some_var([X|L]) :- choice_list(X,CL), !.
some_var([X|L]) :- some_var(L).


one_possibility([ZZZ]).

/* Double-relaxation analysis of a particular value */
/* for a particular variable */
possible_value2(O,CL,V) :- member(V,CL),
  bagof([T,A],constraint(T,A),ConL),
  not(constraint_violation2(V,O,ConL)).


constraint_violation2(V,O,ConL) :-
  twoconstraints(ConL,Type,Args,Type2,Args2),
  member(O,Args), argsoverlap(Args,Args2),
  substitute(O,V,Args,XArgs),
  substitute(O,V,Args2,XArgs2),
  not(double_satisfiable(Type,XArgs,Type2,XArgs2)).


double_satisfiable(Type,Args,Type2,Args2) :-
  constraint_preference(Type2,Type),
  !, double_satisfiable(Type2,Args2,Type,Args).
double_satisfiable(Type,Args,Type2,Args2) :-
  satisfiable(Type,Args,Newargs),
  bind(Args,Newargs,Args2,Result),
  satisfiable(Type2,Result,Newargs2).


twoconstraints([[T,A]|ConL],T,A,T2,A2) :- member([T2,A2],ConL).
twoconstraints([[T,A]|ConL],T1,A1,T2,A2) :-
  twoconstraints(ConL,T1,A1,T2,A2).


argsoverlap(L1,L2) :- member(X,L1), member(X,L2),
  choice_list(X,CL), !.


bind([],[],A2,A2).
bind([V|A],[V|NA],A2,R) :- bind(A,NA,A2,R), !.
bind([O|A],[V|NA],A2,R) :- substitute(O,V,A2,NA2),
  bind(A,NA,NA2,R), !.

/* Utility functions */
repeatactive.
repeatactive :- active(X), repeatactive.


member(X,[X|L]).
member(X,[Y|L]) :- member(X,L).


substitute(X,Y,[],[]).
substitute(X,Y,[X|L],[Y|L2]) :- substitute(X,Y,L,L2), !.
substitute(X,Y,[Z|L],[Z|L2]) :- substitute(X,Y,L,L2).


some_bindings([],[]).
some_bindings([Arg|Args],[Substarg|Substargs]) :-
  choice_list(Arg,CL), !, member(Substarg,CL),
  substitute(Arg,Substarg,Args,Args2),
  some_bindings(Args2,Substargs).
some_bindings([Arg|Args],[Arg|Substargs]) :-
  some_bindings(Args,Substargs).
```

And here's the script of a run on the "send more money" problem, using the problem-dependent file of the last section, without E=5, and plus one additional fact which says sum constraints should be used before unique constraints (since sum requires less computation to check):

```
constraint_preference(sum,unique).
```

Extra blank lines were inserted in the following printout to make it easier to read.

```
?- relax.
Single predicate relaxation begun.
Studying variable e
Studying variable n
Studying variable d
Studying variable o
Possibilities for o reduced to [0,1]
Studying variable r
Studying variable y
Studying variable s
Possibilities for s reduced to [8,9]
Studying variable m
Possibilities for m reduced to [1]
Studying variable c1
Studying variable c10
Studying variable c100

Studying variable s
Studying variable o
Possibilities for o reduced to [0]
Studying variable e
Possibilities for e reduced to [2,3,4,5,6,7,8,9]
Studying variable n
Possibilities for n reduced to [2,3,4,5,6,7,8,9]
Studying variable d
Possibilities for d reduced to [2,3,4,5,6,7,8,9]
Studying variable r
Possibilities for r reduced to [2,3,4,5,6,7,8,9]
Studying variable y
Possibilities for y reduced to [2,3,4,5,6,7,8,9]
Studying variable c1
Studying variable c10
Studying variable c100
Possibilities for c100 reduced to [0]

Studying variable r
Studying variable d
Studying variable n
Studying variable e
Studying variable s
Possibilities for s reduced to [9]
Studying variable c10
Studying variable y
Possibilities for y reduced to [2,3,4,5,6,7,8]
Studying variable r
Possibilities for r reduced to [2,3,4,5,6,7,8]
Studying variable d
Possibilities for d reduced to [2,3,4,5,6,7,8]
Studying variable n
Possibilities for n reduced to [2,3,4,5,6,7,8]
Studying variable e
Possibilities for e reduced to [2,3,4,5,6,7,8]
Studying variable c1
Studying variable c10

Studying variable n
Studying variable d
Studying variable r
Studying variable y
Double predicate relaxation begun.
Studying variable e
Possibilities for e reduced to [2,3,4,5,6,7]
Studying variable n
Possibilities for n reduced to [3,4,5,6,7,8]
```

```
Studying variable d
Studying variable r
Studying variable y
Studying variable c1
Studying variable c10
Possibilities for c10 reduced to [1]

Single predicate relaxation begun.
Studying variable n
Studying variable e
Studying variable d
Studying variable r
Possibilities for r reduced to [3,4,5,6,7,8]
Studying variable y
Studying variable c1
Studying variable n
Studying variable e
Studying variable d

Double predicate relaxation begun.
Studying variable r
Possibilities for r reduced to [8]
Studying variable n
Possibilities for n reduced to [4,5,6,7]
Studying variable e
Possibilities for e reduced to [3,4,5,6]
Studying variable d
Possibilities for d reduced to [2,3,4,7]
Studying variable y
Possibilities for y reduced to [2,3,5,6,7]
Studying variable c1
Possibilities for c1 reduced to [1]

Single predicate relaxation begun.
Studying variable y
Possibilities for y reduced to [2,3]
Studying variable d
Possibilities for d reduced to [7]
Studying variable e
Possibilities for e reduced to [5,6]
Studying variable n
Possibilities for n reduced to [6]
Studying variable e
Possibilities for e reduced to [5]
Studying variable y
Possibilities for y reduced to [2]

choice_list(y,[2]).
choice_list(e,[5]).
choice_list(n,[6]).
choice_list(d,[7]).
choice_list(c1,[1]).
choice_list(r,[8]).
choice_list(c10,[1]).
choice_list(s,[9]).
choice_list(c100,[0]).
choice_list(o,[0]).
choice_list(m,[1]).

yes
```

.SH Keywords:

constraint
label
dependency
dependency-based backtracking
possibility list
relaxation
active variable

Exercises

13-1. (A) Suppose in an expert system written in Prolog we have the following rules for proving that some person X is an **a**:

```
a(X) :- b(X), c(X).
a(X) :- d(X).
a(X) :- e(X).
```

Suppose:

> b is true with probability 0.8;
> c is true with probability 0.7;
> d is true with probability 0.1;
> e is true with probability 0.6;
> b, c, d, and e are probabilistically independent of one another.

Rearrange the three rules, and perhaps the expressions in the first rule, to give the fastest (most efficient) execution of the query

```
?- a(george).
```

13-2. Rearrange the query given in Figure 13-6 for most efficient execution using the suggested rearrangement guidelines. How does regular Prolog interpretation of the rearranged query compare with dependency-based backtracking on the original query?

13-3. (E) For some queries it is better to rearrange the query than to do dependency-based backtracking, but for other queries dependency-based backtracking will always result in fewer backtracks than any rearrangement of the query. Explain why the following exemplifies the latter situation:

```
?- a(X), b(Y), c(Z), d(X,Y), e(X,Z), f(Y,Z).
```

13-4. (A) Suppose we are doing relaxation for picture interpretation. It would be nice if we can determine unique variable assignments when done; that would be strong evidence that we found the correct interpretation of the picture. But what can we conclude if we get instead:

(a) No possible final interpretations?

(b) Two possible final interpretations?

(c) One million possible final interpretations?

13-5. (E) Discuss why relaxation might have more application to artistic creation than any of the other control structures we have discussed in this book.

13-6. (R,A) Budget General Hospital must schedule meals for its patients. Budget tries to save its patients money, so its food is cheap and tastes terrible, and there are only three meals offered:

> Meal m1: gruel
> Meal m2: dehydrated eggs
> Meal m3: leftovers from nearby restaurants

Assist the hospital by scheduling six meals for a patient. The six meals are breakfast, lunch, and dinner on each of the two days. The following constraints apply:

> C1. Breakfast on the first day must be Meal m1, since the patient will still be queasy from their operation.
>
> C2. No two meals in succession may be the same--you don't want the patients to realize how cheap you are.
>
> C3. Meal m3 is the most expensive, so it must only occur once. (It requires a trip to a restaurant.)
>
> C4. Every meal on the second day must have a higher number than the meal at the corresponding time on the first day. For instance, lunch on the first day could be Meal 1 and lunch on the second day Meal m2. (This is because patients are harder to fool as they recover.)

Find a meal plan for the six meals using pure relaxation. After obtaining initial possibility lists, check the possibilities for breakfast on the second day, and proceed from there. Write out each step of your solution, giving constraints used and how they were used.

13-7. (E) Develop a relaxation approach to analyzing pictures like Figure 13-9. This picture was produced by a camera mounted on a robot moving around outdoors. The picture only indicates line boundaries between visual regions of abruptly different brightness. Assume all the lines represent sharp boundaries, and the camera was focussed far away so that everything seen was more than 30 feet away. Assume the camera was oriented so that horizontal is one third of the way down from the top of the picture.

(a) Define ten permissible labels for regions based on their shapes, the ten most reasonable ones you can think of.

(b) List impossible relationships between regions based on these labels (what regions cannot be next to a particular region, etc.) Use **above**, **left-of**, and **inside** for relationships, and any others you think useful. Treat combinations that are possible only rarely as impossible; don't worry about minor exceptions.

(c) Use your constraints to assign as narrow an interpretation as possible to the picture.

13-8. Relaxation can be used to solve an immensely important problem of large organizations: interpretation of acronyms. The full names of many organizations are so long that it's impossible to avoid abbreviating them, yet no one can keep track of all these acronyms. Sophisticated computer programs are maybe necessary.

Suppose for each letter of the alphabet we have a list of words that the letter could represent, in the context of names of government organizations. Suppose:

> N can be national, naval, new, name
> S can be school, staff, science
> C can be center, committee, computer, crazy

Here are the constraints:

> C1: a word at the right end of an acronym must be a noun. The nouns are "name", "science", "school", "staff", "center", "committee", and "computer".
>
> C2: a word anywhere else must be an adjective. The adjectives are "national", "naval", "new", "science", "computer", and "crazy".
>
> C3: the same word cannot occur twice in an acronym.
>
> C4: "national" and "naval" cannot be in the same acronym with "crazy" (no one ever admits something national or naval is crazy, even when it is).
>
> C5: "staff" cannot be in the same acronym with "naval", "national", or "name".
>
> C6: "new" cannot be in the same acronym with "science".

Interpret the acronym "NCSS" using these constraints.

(a) Use relaxation to reduce the possibilities as much as you can. Write out each step. Then write out the possible interpretations remaining.

(b) Suppose we assign probabilities to every word possibility, representing how likely they are to occur in an acronym. So "national" might be 0.5 and "new" might be 0.02 (rarely is anything new in government organizations). Describe a good way to combine these probabilities to choose the "best" one of several interpretations of some acronym, "best" according to a single derived number (give or refer to a formula).

13-9. (H) Try to interpret Figure 13-10 (label the regions) using constraint propagation. You won't be able to find a unique interpretation, but do as much as you can. Initial region labels are as follows:

> The five small regions are either airplanes or ships; anything else is either water, marsh, land, or a cloud.

You have the following constraints:

> C1. Airplane regions are the only ones that can be within cloud regions, but airplane regions can be within any region.
> C2. Ship regions can only be within water regions.
> C3. Two small regions touching one another must be ships.
> C4. A "T" vertex (where three line segments meet, and two of them form an angle of approximately 180 degrees) not involving a ship or airplane means either
>
> > (a) that the region with the 180 degree angle is a cloud, or
> > (b) the region with the 180 degree angle is land and the other two regions are water and marsh.
>
> C5. Every marsh region must have such a T vertex somewhere on its border.
> C6. Water regions cannot border water regions.
> C7. Marsh regions cannot border marsh regions.
> C8. Land regions cannot border land regions.
> C9. The outer (border) region is not a cloud.

Give every step in your relaxation.

13-10.(a) Suppose in relaxation there are ten variables with initially eight possible labels each. Suppose there is a unique solution. What is the size of the search space?

(b) Suppose each step that chooses a variable eliminates half the possibilities for that variable, on the average. What is approximately the average number of steps to reach the unique solution?

13-11. Consider a picture as a graph. Regions border edges, and there are vertices where edges meet. Suppose you do relaxation to interpret the picture, and constraints specify what labels adjacent regions can have. Suppose you draw another graph for which the vertices represent variables in the relaxation, with connections between vertices representing that the two variables both occur together in at least one constraint. What is the relationship of this new graph to the original picture graph?

13-12. (E) Suppose we are certain that there is one and only one solution to a relaxation problem. Suppose when we run this relaxation problem with all the constraints we can think of, we get several hundred possible interpretations. Suppose we can also formulate some "reasonable restrictions", restrictions usually but not always true (their probability being close to 1). Then we can try to get a solution by taking (assuming) some of the "reasonable restrictions" as constraints, and seeing if we get a unique solution to the relaxation. But we want to assume as few restrictions as possible, and we prefer to assume those that are most certain. Explain how finding a unique solution to a relaxation problem by assuming "reasonable restrictions" can be considered a search problem. Describe its characteristics, and recommend an appropriate search strategy.

13-13. (R,A,P) Consider the query:

```
?- n(X), n(Y), n(Z), X>Y, not(f(X,X)), g(X,Y,Z).
```

Suppose the database is:

```
n(1).
n(2).
n(3).
n(4).
n(5).


f(2,2).
f(2,4).
f(3,1).


g(1,3,2).
g(2,3,3).
g(3,3,4).
g(4,3,5).
```

(a) How many times does this query backtrack to get its first answer? Count each right-to-left movement from predicate expression to predicate expression as one backtrack.

(b) Rearrange the query in a form that leads to the fewest backtracks. How many times does it backtrack now? Show the query you used.

(c) Suggest a way to do dependency-based backtracking to answer the original query. Simulate it without the computer. How many times do you backtrack now?

(d) Now answer the original query by doing relaxation without the computer. Assume the standard form of relaxation that tries to satisfy each multivariable constraint separately, throwing away previously found bindings. Write out your steps.

13-14. (A) The demonstration of the pure relaxation program on the cryptarithmetic problem takes a considerable amount of time on the first few variables, then speeds up considerably. If instead of specifying E=5 we say that E is either 0 or 5, the program runs faster and reaches a solution faster.

(a) What accounts for this paradox?

(b) Suggest code that can be eliminated from the definition of this cryptarithmetic problem to significantly lessen or eliminate the paradoxical behavior.

(c) The code that you eliminated in part (b) must have been there for a reason. What was it?

13-15. (P) Suppose you have an eight-digit number whose digits are all different. Suppose that when you multiply this number by its rightmost (least) digit, you get a nine-digit number whose digits are all the leftmost (highest) digit of the original number. Suppose further that the second-highest digit of the original number is 2. Set this problem up in a problem-dependent file to run with the pure relaxation program. Run it and see what you get.

13-16. (H,P) Generalize the "double relaxation" program to "N-relaxation". In other words, take three constraints together if you don't get anywhere with two constraints together, and four constraints together if you don't get anywhere with three, and so on up to the total number of constraints. This will require recursion. The program should degrade to standard Prolog querying in the worst case. Try your program on the cryptarithmetic problem CROSS + ROADS = DANGER.

13-17. (H,P) Besides double relaxation, there is an alternative improvement on pure relaxation: keep possibility lists for pairs, triples, etc. of variables instead of single variables. So you can solve harder problems this way than with pure relaxation, but you need more possibility lists. Write a "pair-relaxation program" that reasons about pairs of values this way.

13-18. (A) We can see relaxation as a search problem with a single operator: picking a variable and a value, and trying to verify that value is still possible.

(a) What is the search space for this problem? Remember, a state for a search problem must include everything about the problem in one bundle.

(b) Which search strategy, of those mentioned in Chapter 9, is most like that used by the algorithm?

(c) Is this search decomposable about intermediate states into simpler subproblems?

(d) How does the branching factor change as search proceeds?

(e) Give an upper bound on the size of the search space.

Go to book index

A more general logic programming

Prolog is a language for *logic programming*. That is, it's a programming language that looks and works somewhat like formal logic. To be efficient and easy to use, Prolog omits some important features of formal logic. We'll discuss now these omissions, and present a more general and powerful--but slower--approach to logic programming. This approach is called *resolution*, and it emphasizes declarative (instead of procedural) meanings of logical formulas.

Logical limitations of Prolog

Prolog can do many things. But it has four fundamental logical weaknesses:

> 1. Prolog doesn't allow "or"d (disjunctive) facts or conclusions--that is, statements that one of several things is true, but you don't know which. For instance, if a light does not come on when we turn on its switch, we can conclude that either the bulb is burned out or the power is off or the light is disconnected.
>
> 2. Prolog doesn't allow "not" (negative) facts or conclusions--that is, direct statements that something is false. For instance, if a light does not come on when we turn on its switch, but another light in the same room comes on when we turn on *its* switch, we can conclude that it is false that there is a power failure.
>
> 3. Prolog doesn't allow most facts or conclusions having existential quantification--that is, statements that there exists some value of a variable, though we don't know what, such that a predicate expression containing it is true. (Prolog does have a limited form of existential quantification for local variables in rules, as discussed in Section 4.1.) For instance, if we know that something is wrong with an appliance, then there exists a component X of the appliance such that X has a fault in it.
>
> 4. Prolog doesn't directly allow *second-order logic*, predicate names as variables--that is, statements about P where P stands for any predicate name. We can get something close with the trick of the **inherits** predicate in the car frame hierarchy of Section 12.11, the idea of rewriting facts to include extra predicate-name arguments. You can also approach second-order logic using the built-in **clause** predicate discussed in Section 7.13. So this weakness of Prolog is less serious than the others, and we won't say anything further about it in this chapter.

Notice that these are logical issues, not efficiency issues. Chapter 6 discussed how Prolog isn't an efficient control structure for reasoning about some problems. But these four points are deeper and more serious weaknesses: they represent things Prolog can't do at all even working slowly.

The logical (declarative) meaning of Prolog rules and facts

To better understand Prolog's limitations and how to get around them, let's examine more carefully what Prolog rules and facts mean.

Chapter 4 explained how rules and facts all go into a Prolog database of true statements. Alternatively, a Prolog database can be seen as a single logical statement representing the conjunction ("and"ing) of the statements of each rule and fact. We'll now show how the declarative or logical meaning (not the same as the procedural meaning) of any Prolog database can be expressed entirely as a set of statements using only "or"s and "not"s.

To do this we must first remove the "**:-**" symbol from rules, because that is not a logic symbol. In Section 4.1 we said to think of it as a backward arrow or backward implication. In logic, an implication (or, strictly speaking, a *material* implication) is equivalent to an "or" (a disjunction, symbolized in Prolog by "**;**") of the pointed-to side of the arrow with the negation of the other side of the arrow. So the rule

```
a :- b.
```

is equivalent in logical (declarative) meaning to

```
a; not(b).
```

which reads as "a or not b". Figure 14-1 shows the truth table for the two forms, to better convince you.

An immediate generalization comes from taking predicate **b** to be an "and" (conjunction) itself. For instance

```
a :-  c, d, e.
```

is equivalent to

```
a; not(c,d,e).
```

which by DeMorgan's Law is the same as:

```
a; not(c); not(d); not(e).
```

So we can express the logical equivalent of any Prolog rule by "or"ing the left side with the **not**s of each expression that is "and"ed on the original right side. This is called the *clause form* of the rule.

But logical equivalence is not complete equivalence of meaning because it only covers the declarative meaning of rules. Prolog rules also have the "procedural" interpretation of "If you believe that these things hold, then believe that this thing holds". So rules involve a causation, a moving from a cause (a belief in several things) to an effect (a belief in something else); and the effect comes after the cause in time. Causations imply a direction of reasoning, while an "or" doesn't necessarily. For instance,

```
a :- b.
```

models a causation from **b** to **a**. But its logical equivalent in clause form

```
a; not(b).
```

can also be used from **a** to **b**: if we are told **a** is false, then **b** must be false too so the "or" will be true. That's the *contrapositive* direction of reasoning, and Prolog rules can't be used that way even though it's logically valid | REFERENCE 1|. .FS | REFERENCE 1| Don't confuse the backward reasoning of the contrapositive with backward chaining. Backward chaining reasons about the left side of some rule, whereas contrapositive reasoning reasons about the *opposite* of the left side. .FE

But clause form has advantages too. The second limitation on reasoning of Prolog interpreters mentioned in Section 14.1 concerned **not**s: they mean "impossible to prove", not "proved false". But when we write things in clause form we can interpret **not**s more precisely, to reach new conclusions not otherwise possible. For instance, in the previous contrapositive reasoning example, when **a** is false, **b** is proved false, not just proved to be impossible to succeed. So clause form can provide "true nots". But there is an associated danger: when the Prolog rule itself has a **not**, we must make sure that we can interpret it as provably false in the real world. Otherwise, the clause form only covers part of its meaning.

Extending Prolog rules

Using the preceding clause-form conversion for Prolog rules lets us give meaning to new kinds of rules, rules not legal in Prolog. For instance this "pseudo-Prolog"

```
(a; b) :- c.
```

which means that either of **a** or **b** is true whenever **c** is true, becomes in clause form

```
a; b; not(c).
```

And this pseudo-Prolog

```
not(a) :- b.
```

which means **a** is false whenever **b** is true, becomes

```
not(a); not(b).
```

Notice that the first clause-form formula has two unnegated expressions, and the second has no unnegated expressions. In general, any Prolog rule without **not**s becomes a clause form having one and only one unnegated expression, what's called a *Horn clause*.

Clause form for a rule can require more than one "or"ed formula. As a more complicated example, consider this pseudo-Prolog

```
(a; (b, c)) :- d, not(e).
```

which has the logical equivalent

```
a; (b, c); not(d); e.
```

To get rid of the "and", we can use the distributive law for "and" over "or". This gives two separate statements (*clauses*), each of which must be true:

```
a; b; not(d); e.
a; c; not(d); e.
```

And that's the clause form for the original rule.

Rewriting rules in clause form answers some puzzling questions of why rules sometimes seem "and"ed together and other times "or"ed together. Suppose we have two rules

```
a :- b.
a :- c.
```

The logical equivalent form is

```
(a; not(b)), (a; not(c)).
```

or:

```
a; (not(b), not(c)).
```

using the distributive law of "and" over "or". This can be rewritten as a single rule

```
a :- (b;c).
```

using DeMorgan's Law. So an "and" in the one sense--the "and" of the logical truth of separate rules--is an "or" in another--the "or" of the right sides of rules with the same left side.

More about clause form

So a clause is an "or" of a bunch of things, each of which is either a single predicate expression or the **not** of one. As usual, expressions can have arguments. For instance, this "santa clause":

```
santa(joe); santa(tom); not(santa(bill)).
```

Any statement we can express in first-order logic has a logical equivalent in a set of clauses. Why is this important? Because there's a simple yet powerful inference method that can be applied to clauses, *resolution*. It can be proved that *any* inference that logically follows from a set of statements can be found by using resolution on the clause forms of those statements. So resolution is more powerful than Prolog-style backward chaining.

But to use resolution, everything must be in clause form. We've already illustrated a procedure for translating "pseudo-Prolog" with "or"s, "not"s, and arbitrarily complex formulas on the left sides of rules: just rewrite in *disjunctive normal form* as an "or" of "and"s, using the laws of logic. That covers the first two limitations of Prolog cited in Section 14.1. But what about existential quantifiers? They can get complicated.

The simplest case for existential quantifiers is when we want to assert there exists some variable value such that a predicate expression mentioning that variable is true. Then we can substitute a constant for the variable, provided that constant can't be confused with an actual value of the variable, like a nonsense word. For instance, if we want to say that there exists an X such that **p(X)** is true, then we can could assert fact **p(zzxxy)** provided that **zzxxy** is not a symbol for any of the actual values for X. We can then use this fact **p(zzxxy)** in chains of reasoning, just remembering that this value doesn't really mean anything.

But now suppose we want to say that for every Y there exists X such that **p(X,Y)** holds. Now we can't just substitute a constant for X because X may depend on Y: that is, X is a function of Y. This function is a *Skolem function*. We need one whenever we are trying to represent in clause form a statement containing both existential and universal quantification. Standard techniques exist for situations needing Skolem functions, but they are too complicated to discuss here. See books on "theorem proving" if you're interested.

Resolution

Resolution is an inference technique that takes two clauses as input, and produces a clause as output. The output clause, the *resolvent*, represents a true statement consistent with the input clauses, the result of *resolving* them. In other words, the resolvent is one conclusion we can draw. If the resolvent is a fact, then we've proved a fact. If the resolvent is the clause consisting of no expressions, the *null clause*, we've proved a contradiction. Resolution is particularly efficient for proof by contradiction: we assume the opposite of some statement we wish to prove, and see if we can prove the null clause from it.

Resolution requires pairs of opposites in the two input clauses. That is, one input clause must contain a predicate expression--call it P--for which **not(Q)** occurs in the other input clause and where P can match Q by binding variables as necessary. (Formally, P matches Q if the expression **P=Q** can succeed.) Then the resolvent of the two input clauses is the "or" of everything besides P and **not(Q)** in the two clauses, eliminating any duplicate expressions. We say that the P and the **not(Q)** "cancel". For instance, if the input clauses are

```
a; b; not(c); d.
e; not(b); a; f.
```

then the resolvent (output) clause is

```
a; not(c); d; e; f.
```

where we eliminated the opposites **b** and **not(b)** and a duplicate **a** fact.

Inference by resolution becomes most useful when we do several resolutions in succession. Here's an example. Let's use Horn clauses (clauses with one and only one unnegated expression), because we already know how Prolog interpreters handle them, and let's avoid variables. Suppose we have these rules and facts:

```
a :- b, c.
c :- d.
b.
d.
```

Rewriting in clause form, we get these Horn clauses:

```
a; not(b); not(c).
c; not(d).
b.
d.
```

(Prolog facts are identical in clause form.) Now suppose we want to prove **a** (see Figure 14-2):

1. To do proof by contradiction, we add **not(a)** to the other four clauses in our database of true statements.

2. Resolving **not(a)** with the first of the four original statements, the **a** and **not(a)** expressions cancel, and the resolvent is **not(b); not(c)**. We can add that new clause to the others.

3. Resolving this new clause with the third of the original clauses, the **b** and **not(b)** cancel, and the resolvent is just **not(c)**.

4. Resolving this in turn with a second of the original four clauses, we cancel the **c** and **not(c)**, giving as resolvent **not(d)**.

5. Finally we resolve this with the last of the original four clauses, and **d** and **not(d)** cancel, leaving us with a null clause.

6. Therefore we can prove anything if we assume that **a** is false. So **a** must be true. (This assumes that the original set of rules and facts was not self-contradictory, something we could verify by doing all possible resolutions among them.)

There is a one-to-one correspondence of the steps in the previous *resolution proof* and the steps that Prolog interpreters follow in backward chaining. To prove **a**, they would:

1. Take **a** as the goal (query).

2. Find a rule for **a**: the first rule. This says to prove **b**, then **c**.

3. But **b** is a fact (the first fact).

4. To prove **c**, use the second rule, which says to prove **d**.

5. But **d** is a fact (the second fact).

6. Therefore **c** is true and **a** is true.

In general, resolution can do anything backward chaining can do, but not the other way around. Resolution is a more general and flexible form of inference, because it can resolve clauses in many different orders; backward chaining is more rigid. And every new resolvent clause can be used many ways in new resolutions, so resolution possibilities keep increasing--it's a *monotonic* search process.

Resolution with variables

When predicates have variable arguments, resolution becomes a little more complicated: we still look for a pair of opposites, but Prolog-style binding of the variables can be done to make the canceling expressions "match". As with Prolog, bindings made to variables apply to any other occurrences of the variables within their original clauses, so if a **p(X)** in the first input clause matches a **p(4)** in the second input clause, any other **X** in the first clause becomes a 4. Variables can also be bound to other variables. Important note: it's essential that each input clause have different variable names before resolving.

Here's an example of resolution with variables. Suppose the two clauses are

```
a(3); b(Y); not(c(Z,Y)).
not(a(W)); b(dog); c(W,cat).
```

The **a** expressions can cancel with **W** bound to 3, giving:

```
b(Y); not(c(Z,Y)); b(dog); c(3,cat).
```

The **b(dog)** is redundant with **b(Y)**, so we can improve this clause to:

```
b(Y); not(c(Z,Y)); c(3,cat).
```

But we could resolve the original two clauses another way. The **c** expressions could cancel, with **Z** being bound to **W** and with **Y** being bound to **cat**, giving:

```
a(3); b(cat); not(a(W)); b(dog).
```

This is a completely different resolvent, representing a different conclusion possible from the two clauses. Notice that we can't eliminate anything here; **b(cat)** and **b(dog)** aren't redundant, nor are **a(3)** and **not(a(W))**.

Note that bindings are transitive: if **A** is bound to 9, and **B** is bound to **A**, then **B** is bound to 9 too. So several reasoning steps may be necessary to determine a variable binding.

Three important applications of resolution

Resolution is a powerful inference technique that can supplant other inference techniques. Three important special cases of resolution are summarized in Figure 14-3: backward chaining, forward chaining, and rule collapsing. To use resolution for backward chaining as in Section 14.5, one starting clause is always the negation (opposite) of something to be proved. To use resolution for forward chaining, one input clause for every resolution is always a fact. Rule collapsing is a way to make rules more efficient, not a control structure itself: it takes two rules, one whose left side is on the other's right side, and combines them into a single new rule; this is equivalent to resolving the rules' clauses. Procedure collapsing is important to compilers for programming languages.

Resolution search strategies

Reasoning by resolution means performing a series of resolution operations. This often means more things to choose from than conflict resolution in rule-based systems, because you must pick pairs of clauses to resolve, and typically there are many. Facts are clauses too, as are the resolvents (results) from past resolutions. So resolution-based reasoning is a kind of search with a high branching factor. This means a breadth-first resolution strategy (resolving every pair of original clauses, then every new clause with either an original clause or a new clause, and so on) is very slow. In principle, a breadth-first resolution control structure can prove anything that is a logical consequence of particular facts and rules; but that's no good if it takes a hundred years to do so.

Often we know what we want to prove. Then we can use a *set-of-support* strategy, which we used without saying so in the example of Section 14.5. The idea is to assume the opposite of what we want to prove, and resolve it repeatedly with other clauses until we reach the null clause. If no possible resolutions remain at some point (when no other clause has an opposite that can "cancel"), back up to the last previous resulting clause for which there were alternative resolutions, and take an alternative. This is basically a depth-first resolution strategy starting from the negation of the proof objective.

If we don't have any one thing in particular we want to prove, but we prefer to prove facts, then a *unit-preference* resolution strategy may be good. The idea is to do first all resolutions involving facts, both positive and negative. If there aren't any facts, then perhaps prefer resolutions involving two-argument clauses, and so on. This strategy tends to keep resolvent clauses short, which often means we discover new facts fast.

If our clauses all represent rules, we may want to do the rule *collapsing* mentioned in the last section. A breadth-first resolution strategy could work for this, since there's no designated objective or facts to work from. For efficiency, we might try instead best-first search with the evaluation function the total number of expressions in the input clauses, which tends to discourage less useful resolutions. In any event, we should arbitrarily stop the search at some point, since there can be enormous numbers of possible resolutions, and we don't want to try them all.

Domain-dependent heuristics can help considerably when reasoning with resolution. Heuristics can enhance any of the search strategies mentioned.

Implementing resolution without variables (*)

Resolution without variables is simple to implement in Prolog. Represent the clauses as list arguments to **clause** facts, so for instance

```
a; not(b); not(c); d.
```

is represented as

```
clause([a,not(b),not(c),d]).
```

Notice lists require commas, so the comma here actually means "or". Then query the predicate **go**. Everything new proved will be printed out. Here's the program:

```
/* Resolution without variables */
go :- resolution(C1,C2,Cnew), !, write(Cnew), nl,
  not(Cnew=[]), go.


resolution(C1,C2,Cnew) :- clause(C1), clause(C2), not(C1=C2),
  matched_items(C1,C2,C1item,C2item), delete(C1item,C1,C1d),
  delete(C2item,C2,C2d), union(C1d,C2d,Cnew), not(clause(Cnew)),
  not(tautology(Cnew)), not(some_superset(Cnew)),
  asserta(clause(Cnew)).


matched_items(C1,C2,C1item,not(C1item)) :- member(C1item,C1),
  member(not(C1item),C2).
matched_items(C1,C2,not(C2item),C2item) :-
  member(not(C2item),C1), member(C2item,C2).


some_superset(C) :- clause(C2), subset(C2,C).


tautology(C) :- member(X,C), member(not(X),C).
```

Notice that we check new clauses to make sure they're neither previously found, nor tautologies (always-true statements), nor immediately derivable from other clauses by removal of items; only then do we assert a new clause. The assertion uses **asserta**, so a depth-first strategy much like set-of-support will be used.

This requires the **member**, **delete**, and **subset** predicate definitions from Sections 5.5, 5.6, and 5.7 respectively, plus the **union** predicate from Section 11.4 (which is closely related to the **append** of Section 5.6). Using **union** instead of **append** prevents repeated expressions in the resolvent.

```
/* Utility functions for resolution */
```

```
member(X,[X|L]).
member(X,[Y|L]) :- member(X,L).


union([],L,L).
union([X|L],L2,L3) :- member(X,L2), !, union(L,L2,L3).
union([X|L],L2,[X|L3]) :- union(L,L2,L3).


delete(X,[],[]).
delete(X,[X|L],M) :- !, delete(X,L,M).
delete(X,[Y|L],[Y|M]) :- delete(X,L,M).


subset([],L).
subset([X|L1],L2) :- member(X,L2), subset(L1,L2).
```

For a demonstration, suppose we have these clauses:

```
clause([a,not(b),not(c)]).
clause([b]).
clause([d,not(a)]).
clause([c,e,f]).
```

Here's what happens when we run the program:

```
?- go.
[a,not(c)]
[not(c),d]
[d,e,f]
[a,e,f]


no
```

Unfortunately, implementing resolution with variables in Prolog is very tricky. The problem is in handling redundancies, the **tautology** and **subset** predicates of the program, for which Prolog's normal variable-binding must be subverted. Such full resolution inferencers can be written more easily in more flexible programming languages like Lisp.

.SH Keywords:

logic programming
disjunctive fact
negative fact
existential quantifier
second-order logic
clause form
Horn clause
Skolem function
resolution
breadth-first resolution strategy
set-of-support resolution strategy
unit-preference resolution strategy

Exercises

14-1. (A) Resolve all possible ways and list bindings:

```
state(3,A,X); possible(X,X,A).
not(possible(5,Y,Z)); state(3,6,Y).
```

14-2. (R,A) Suppose you are working for a research organization. Suppose you can get travel money if your department chairman approves and your sponsor approves. Alternatively, you can get travel money if your department chairman approves, the boss over him or her approves, and there are discretionary department funds available.

(a) Represent the preceding as two Prolog rules. (Hint: use one-letter predicate names because you'll have to write them many times in this problem.)

(b) Represent the two rules in clause form.

(c) Suppose that these are the only two ways that you can get travel money. Therefore if you do get travel money, certain things must have been true. Write this implication in "pseudo-Prolog" as a rule with the symbol "**:-**", but with "and"s, "or"s, and "not"s on its left side. Your rule must cover *all* the implications of having travel money.

(d) Convert this pseudo-Prolog rule to clause form. (Hint: clause form here is three clauses.)

(e) Suppose you can get travel money. What new clauses logically follow from this? (Make sure there are no unnecessary extra expressions in the clauses.)

(f) Suppose you can't get travel money. What new clauses logically follow from this? (Make sure there are no unnecessary extra expressions in the clauses.)

14-3. (a) Represent the following in clause form:

C1: Block A is on the table.
C2: Block B is on block A.
C3: Block C is on block A.
C4: Block D is on block C.
C5: Block A is blue.
C6: Block B is blue.
C7: Block C is red.
C8: Block D is green.
C9: A block is above another block X if it is on that block or else if it is on a block which is above X.

(b) Prove by resolution that block D is above a blue block. Give numbers of statements you resolve, and label your resolvents.

14-4. Suppose we're told to resolve the clauses

```
a; b.
not(a); not(b).
```

Can we simultaneously cancel out both matched pairs, getting the null clause as resolvent? Why or why not?

14-5. (R,A) By the definition of resolution given in this chapter, if we resolve the clauses

```
a; b.
not(a); c.
```

we get

```
b; c.
```

But this doesn't seem to make sense if we show what's happening in a Venn diagram (See Figure 14-4). Here the region marked with lines running from southwest to northeast represents the first clause, and the region marked with lines running southeast to northwest represents the second clause. Any pair of clauses that are each individually true can be considered to be "and"ed together. But the region that has both markings (the cross-hatched region) does not correspond to the preceding resolvent clause. What's wrong?

14-6. (E) Suppose we have these Prolog rules:

```
a :- b, c.
a :- not(b), d.
```

Suppose that **not** can also be interpreted as a "real" not, one that insists that negative evidence is present.

(a) Write the two rules in clause form.

(b) Now resolve the two clauses from part (a), and write a new Prolog rule without **not**s equivalent to the resolvent clause.

(c) Explain how the idea of the preceding could be generalized to a useful trick to improve rule-based expert systems, one that applies to rules of any length. Explain (i) how you pick appropriate (good) pairs of rules, (ii) how you can avoid converting to clause form, and (iii) under what circumstances you can delete the original rules.

14-7. Prolog interpreters are Horn-clause theorem-proving systems. But consider the following fallacy.

(a) Represent in clause form this pseudo-Prolog:

```
(weekend(T); holiday(T)) :- empty(spanagel421,T).
```

which means that if Spanagel 421 is empty, it must either be a weekend or a holiday.

(b) Convert this into an equivalent Prolog rule with only one expression, **holiday(T)**, on its left side.

(c) Show that the original rule and the rule for part (b) have equivalent truth value by showing their truth tables.

(d) Part (b) is a legal Prolog rule because it has only one predicate expression on its left side. So it seems we can always implement a non-Horn clause in Prolog. What's wrong with this argument?

14-8. (E) Suppose we wish Prolog interpreters to use mathematical induction proof techniques automatically. Discuss what is wrong with just adding an induction rule to all Prolog programs, saying in essence

```
"p(X) is true if p(1) is true and if for N>1, p(N) implies p(N+1)."
```

14-9. (A) Consider proving things by resolution as a search problem with one operator: resolve two clauses to get a new clause. Each state can be described as a set of clauses given or proved. Suppose you have N clauses to begin.

(a) What is an upper bound on the initial branching factor?

(b) What is an upper bound on the branching factor after K resolutions have been done?

14-10. (P) The resolution program can be made more efficient.

(a) Improve it so it doesn't examine every pair of clauses, but only pairs in which the first clause precedes the second in the database.

(b) Improve it to remove any clause that *becomes* redundant, in that the expressions of the last clause discovered are a subset of the expressions of that clause.

Go to book index

Testing and debugging of artificial intelligence programs

Artificial intelligence programs tend to be big, complicated programs. (If not in this book, in practical applications.) So preparing for good testing and debugging is especially important. Standard software engineering techniques for managing large programs (modularity, top-down decomposition, good naming, separate testing of components, and so on) help, but certain other less common techniques are particularly important to artificial intelligence. These are summarized in Figure 15-1.

The gold standard

Artificial intelligence programs try to show intelligent behavior. But as we said in Chapter 1, "intelligence" is a vague word. So it's often hard to see whether an artificial-intelligence program is working properly. Of course, there are obvious mistakes, like a car repair program that concludes your engine is destroyed when the problem is a flat tire. But the less obvious mistakes are more dangerous.

What is usually done is to define an absolute or *gold* standard of behavior against which an artificial-intelligence program can be measured. Often (as for expert systems) a program tries to automate things done by certain humans, so we can just ask these humans whether the program is doing things right. For instance, for auto repair, there are master mechanics; for medical diagnosis, there are doctors that are authorities in their subfield; for scheduling, there are people that schedule all the time, like some secretaries; for finding routes in a city, there are experienced taxi drivers; for solving puzzles, there are expert puzzlers; and for mellowing out, there are California residents. Sometimes there is no single human expert, but different people are expert on different pieces of a problem and we can pool their expertise, as often case with knowledge of how to best accomplish things in a bureaucracy.

Unfortunately humans often hate computers, particularly when they are being replaced by computers without compensating rewards, so we must be extremely careful to make the comparison of computer with human behavior a fair one. It's important to give the humans and the computers the same problems to work on (perhaps not telling the humans that they will be compared to computers and not telling the computers that they will be compared to humans, though computers tend be more broad-minded). Then we compare results and mark any discrepancy as wrong. If our graders don't know themselves which results were by computers and which by humans, only some supervisors then we have a *double-blind* test, a good form of testing often used in medical research.

Cases

Once a gold standard is established, it is important to test an artificial intelligence system many ways, or against many situations. These situations are called *cases* or *test cases*; a set of cases used for testing is called a *corpus*. The more complicated the program, the more cases we should run. It is important to test a representative spectrum of cases. It is desirable to have at least one test case for each major program behavior, for each significant combination of inputs, and for each significant kind of output. Since this isn't always possible, judgment is required to reach a compromise.

Focusing on bugs

Testing performance on a set of cases creates two categories: cases for which the standard and the program agree, and cases for which they disagree. The second category is the more interesting one. We can examine these program "failures", and try to identify characteristics that distinguish them from "successes".

Several tools can help. A very important one is trace information, the record of reasoning done about a case. Rules that appear often in failure cases are suspect. Search problems provide partial traces in their operator-sequence output. But expert systems usually just give a single answer, and can be considerably enhanced with a trace of what rules gave the answer.

Other useful information is provided by a confusion matrix, something especially helpful with rule-based systems. It summarizes not only when the program went wrong, but how it went wrong. An example is given in Figure 15-2. Program results were grouped into categories and cross-tabulated with the gold standard. Rows represent gold-standard categories and columns represent program-result categories. The entry in row i and column j is the number of times the program gave an answer in category j when the standard gave an answer in category i. (We assume that one of these diagnoses is a default, so the program always reaches one of these four diagnoses.) So a confusion matrix shows which categories tend to be confused with which other categories, and suggests where rule improvements are needed. For instance according to Figure 15-2, the "electrical short" problem is the most frequently misdiagnosed (only 29 out of 55 cases were correctly recognized), suggesting that more rules for electrical shorts or rules with less restrictive conditions are needed. On the other hand, "fuel blockage" is the diagnosis made most frequently misapplied (only 25 out of 45 fuel blockage diagnoses were correct), suggesting that more and stronger conditions in its rules are necessary to filter out mistakes.

Exploiting pairs of similar cases

Pairs of very similar cases are helpful in testing and debugging of artificial-intelligence programs. They can localize problems and provide generalizations.

A particularly interesting case pair is when the standard and the program disagree on one case, but agree on a very similar case. In other words, whenever failing cases are *near misses* to success. Then the reason for the failure must lie in the small difference in characteristics of the two cases, which helps focus on the source of the bug. For example, suppose we have in a rule-based system:

```
a :- b, c.
```

Now suppose we test (among others) two cases: case X in which facts **b** and **c** are true, and case Y in which **b**, **c**, and **d** are true. Suppose X is a near miss for conclusion **a**, in that the program (or specifically, this rule) concludes that **a** is true while the gold standard says that **a** is false. The program will also conclude that **a** holds in case Y, and assume furthermore that the standard agrees. Since facts **b** and **c** are present in both X and Y, we can guess that the **d** fact makes the difference. There are other explanations, but that seems the simplest. So to get a better rule-based system, we should rewrite the rule as

```
a :- b, c, d.
```

In other words, we figured the original rule was too general, so we fixed it to apply to only one of the two original cases.

Pairs of similar cases can also suggest rule generalizations, a kind of *learning*. Suppose we have in a rule-based system:

```
a :- b, c, d.
```

Suppose case Y is as before, with **b**, **c**, and **d** true. Suppose in case Z that **b** and **d** are true but **c** is false. Suppose the gold standard says that Y and Z are both correct cases for conclusion **a**. Then we can guess that the **c** doesn't matter, suggesting the rule:

```
a :- b, d.
```

This is a riskier change than the previous "near-miss" one, because we are generalizing a rule to apply to more cases. But it's often reasonable.

When the preceding generalization is too strong, we can make weaker changes too. For the appliance application of Chapter 7, we might have written a rule:

```
diagnosis('short in cord') :- askif(device_dead), askif(cord_frayed).
```

Suppose we are now told that when there is a frayed cord and the lights just went out, a "short in cord" diagnosis should also be made. This might suggest

```
diagnosis('short in cord') :- askif(cord_frayed).
```

But that's not a smart idea, because a frayed cord is only a weak symptom and can also suggest a break in the cord (that is, a nonconducting cord). Instead, we should use a concept covering both dead devices and the lights going out: a fuse being blown. That suggests:

```
diagnosis('short in cord') :- diagnosis('fuse blown'), askif(cord_frayed).
```

Figure 15-3 tabulates some inferences of this sort. Current research is extending these ideas to build rule-based systems automatically from examples and nonexamples (that is, descriptions of situations that are definitely not examples of particular concepts)) defining appropriate behavior. This is part of the subfield of *automatic programming* within artificial intelligence. The issues are hard--just think how hard it is for you yourself to write a program.

Composite results

If running an artificial intelligence program gives a result with components, then evaluating the difference between the program and the gold standard is a little more complicated. If the pieces of the result are independent, then the program and standard can be compared separately for each piece, as for a relaxation application in which the result is separate possibility lists for variables. If pieces of the result are linked in a data structure, then structures must be matched to establish corresponding components before comparison, as for a search application in which we must match program operators with the gold standard operators in corresponding order. In either case, only the pieces that do not match need be studied further.

Here's an example. Suppose for some search problem the gold standard finds the operator sequence

```
[a,b,c,d,e,f,g,h]
```

while the program result is

```
[a,b,r,s,d,c,e,f,h]
```

Three things seem to be different in the program result: additional operators **r** and **s** have been inserted after action **b**, operators **c** and **d** have been interchanged, and operator **g** has been omitted. We can thus focus our attention on the parts of the search program that affect those things. So we should check the conditions controlling selection of **r** and **s** to see why they were erroneously fulfilled, and check the conditions on **g** to see why it was erroneously omitted; the switch of **c** and **d** suggests a mistake in the priority of those two operators.

Numbers in comparisons

Numbers--probabilities, evaluation functions, and cost functions--complicate comparisons of a program and a gold standard. There are three ways to handle them. First, we can use the numbers to pick a "best" program result, and compare this to the gold standard. For a diagnosis expert system that assigns probabilities to diagnoses, we can take the diagnosis with the highest probability; for a search program that finds paths to a goal, we can take the lowest-cost path. Second, we can treat the numbered items as a composite structure as discussed in the last section. For an expert system that concludes that diagnoses **a**, **b**, **c**, and **d** are probable with probabilities 0.9, 0.7, 0.6, and 0.3 respectively, while a gold standard rates them 0.8, 0.5, 0.6, and 0.2 respectively, we can focus the debugging **b** and **c** because their numbers are out of order. Third, we can do statistical analyses; for instance, we can take the sum of the square of the deviations of the program-result numbers from the gold-standard numbers to measure closeness of the program and the standard. We can then quantify any proposed program improvement by this criterion. So it can be used as an evaluation function in a "search" for a better program.

Preventive measures

By current estimates an ounce of prevention is worth 1.07 pounds of fixes, so it's important that an artificial-intelligence programming environment provide software tools to help programmers avoid mistakes in the first place. One big help is predefined expectations about the format of code, as forms or frames in the style of Chapter 12. For rule-based expert systems we can use rule frames, to capture the frequently strong structural similarities between different rules. For instance, rules in a car-repair diagnosis system usually go like this:

> (1) localize the problem in a part of the car;
> (2) ask about easily observable symptoms;
> (3) if necessary, ask about symptoms that take a little work to establish (like opening up an object);
> (4) if necessary, ask about experimental actions that need to be done to confirm the diagnosis.

For example:

```
diagnosis('Dead battery') :-
  electrical_problem, lights_dead, not(battery_corroded), battery_voltage_low.
```

A programmer writing a new car-diagnosis rule can be provided with this list of four things as a guide, or warned when departing from this format. Or to be more ambitious, a system could automatically find analogous rules to a new rule a programmer wants to write. Inheritance on a hierarchy of rule-class frames is helpful for this.

Supporting intuitive debugging

A gold standard of performance for an artificial intelligence program can be hard to get and costly to use. Early in debugging, a more intuitive approach to performance analysis using *explanation* facilities is better, and this is common in artificial-intelligence development software like expert systems *packages* and *shells*. After running a program, these facilities can report on what your program did, how it reached a particular conclusion, or why it considered a particular line of reasoning and not another. Such explanations can be helpful even when the system is fully debugged; they help people understand the system better, use it more effectively, and have more confidence in its results.

The commonest explanation capabilities are for:

> "How" questions: questions as to how the program reached a particular final or intermediate conclusion. When the reasoning is simple, a printout of every step will suffice--just the trace information mentioned in Section 15.3. But when reasoning is complicated, this is too much information. For instance, Prolog trace utilities (see Appendix D.12) generally print out information not only on calling a rule and succeeding with it, but on failing and on backtracking to it. (Try tracing the appliance diagnosis program in Section 7.6 or the flashlight diagnosis problem in Section 11.5, if you have tracing facilities in your Prolog dialect, to see the hundreds of reasoning steps done.) So partial summary of trace information is usually provided in artificial-intelligence environments and packages. For instance, a "how" explanation can list just the "top-level" predicates used to reach a conclusion, and then provide the programmer additional details on demand.
>
> "How not" questions: questions as to why something was *not* concluded. These arise when expectations are violated, and there are two very different situations. First, reasoning found the conclusion false; then the trace information answers the question. Second, no try was made to prove the conclusion; then the program control structure must be studied to determine why different lines of reasoning were pursued instead. For instance, the only rules that could have established the conclusion may occur beyond the point where processing went in a sequence of rules.
>
> "Why ask" questions: questions as to what purpose a question to the user serves. These are only appropriate in expert systems that question a user to get data, like the appliance repair program in Chapter 7 with its **ask** predicate. Instead of answering a question, a user might reply "Why ask that?" and get a description of a higher-level issue the question will help resolve or a description of what the program is working on. As with "how" questions, summarization helps. That is, a user doesn't usually want just the issue (rule left side) immediately responsible for the question but the higher-level issues too.
>
> "Why better" questions: questions as to why one alternative was judged better than another, when alternatives are ranked by numbers (probabilities, evaluation functions, or cost functions). These are hard to answer because they depend on mathematical formulas as well as logical reasoning. One approach is to take the trace record ("how" information) from two comparable conclusions, eliminate everything identical, and try to find one-to-one correspondences between the remaining

subconclusions. Explanation can focus on the relative desirability of corresponding subconclusions.

The hope in providing such information to programmers and users is to permit them to use their own intuitions on cases, comparing their reasoning with what the program did. Often artificial intelligence programs try to formalize human intuition, especially expert systems, so an error by a program can "leap out" at the human studying its results.

Evaluating cooperativeness

Right answers are not the only criterion we have for evaluating an artificial-intelligence program. We may also care how friendly or cooperative it is, especially if it requires a lot of human interaction. Tools such as questionnaires for measuring this are common in organizations that do considerable software development. Some issues to consider:

--Is the output comprehensible?
--Does the program use specialized jargon?
--Can the program paraphrase its questions to the user?
--Can the program explain its reasoning well?
--Does the program catch obvious user mistakes?
--Does the program guess well what the user meant in a mistake?
--Does the program take what the user says too literally?

On problems unsuitable for artificial intelligence

The performance of a program may still be unsatisfactory after a long period of testing and debugging. Despite all the clever things that artificial intelligence techniques can do, they can't solve everything. Sometimes it is hard to know in advance whether a problem is suitable. But some guidelines can be given.

First, a problem suitable for artificial intelligence techniques must be sufficiently well-defined to have clear criteria for success and failure, just as for any other technical method. Otherwise, you can't tell when it's working, and you can't easily test and debug it. For instance, an expert system could create works of art, but it would be hard to evaluate--there's a lot of disagreement between art "experts".

Second, the desired program behavior must be sufficiently stable to permit cost-effective testing and debugging. For instance, an expert system to recommend stock-market investments would be hard to build because investment opportunities are constantly changing ("buy low, sell high" is too general to be useful).

Third, it must be possible to use the program routinely. This is obvious but often overlooked. For instance, most military "battle management" systems are a bad idea because it is too hard to keep them correctly and quickly informed of everything that is happening in a battle (especially with communications equipment itself under attack), though prototype implementations for artificially constructed scenarios might seem to work well.

Fourth, an artificial intelligence system must not be reductionist in a dangerous way. As we mentioned in Chapter 1, reductionism is the extent to which an artificial intelligence program fails to capture subtleties of human behavior it is trying to imitate. A highly reductionist program may be safe when its users thoroughly understand its limitations, but can be very dangerous otherwise. For instance, a significantly reductionist battle management system could order troops to fire on themselves, or be vulnerable to sabotage in surprising ways.

Fifth, artificial intelligence techniques should generally not be used when processing time, storage space, or calculation accuracy is critical. The techniques and programs are complicated and often probabilistic or heuristic. A working artificial-intelligence program can often be optimized or compiled to run faster, take up less space, or make more accurate conclusions, but the main idea of artificial intelligence is to provide "intelligence" in computers first, efficiency later. So for instance a "robot warplane", an autonomous military aircraft controlled by real-time artificial-intelligence software, is not likely to work fast enough because things happen too fast in a battle, a large computer imposes a weight penalty on the aircraft, and high accuracy in maneuvers is essential.

Sixth, an artificial intelligence program must be sufficiently complicated, but not too complicated, so that artificial intelligence techniques do better than competing approaches. For instance, in spite of Chapter 11, it's a poor idea to write a search program to fix flashlights because people can figure it out themselves. But an expert system to replace all doctors isn't reasonable, because there are millions of diseases and millions of symptoms, and just the rule management and indexing problem for such a huge expert system would be mammoth. And, in the United States anyway, a flawed system could be sued for malpractice.

You must study carefully these criteria before writing an artificial-intelligence program. The criteria are on the borderline between the technical and the nontechnical. Should this discourage a programmer from worrying about them? No, not at all. Too often technical people ignore such issues when technical insights about them could be invaluable. Insights about artificial intelligence are particularly important today because so few people understand the field--it's hard to learn, as we said in Chapter 1. But as a new area of research compared to other fields of engineering, artificial intelligence has exciting promises for the future, promises generating considerable enthusiasm. Now that you have read this book you can help fulfill them. Use your knowledge wisely.

.SH Keywords:

```
case
gold standard
trace information
confusion matrix
near miss
automatic programming
explanation facilities
reductionist
```

Exercises

15-1. (A) Give a better gold standard for auto repair than an opinion of an expert mechanic.

15-2. Suppose we have a backward-chaining rule-based diagnosis system written as Prolog rules. How could we modify the rules to automatically provide a trace of rules used along with every diagnosis?

15-3. (A) Suppose we have a rule-based system like the appliance diagnosis program in Chapter 7 for which the order of the rules makes a considerable difference in what conclusion is reached first. Suppose for some diagnosis rule-based system, we have the diagnosis rules for two diagnoses in the wrong order. How could a confusion matrix clue us that this is the case?

15-4. Suppose you wanted to construct a rule-based system automatically from examples and nonexamples of its desired behavior, using the ideas of Figure 15-3. The criteria in the Figure only refer to rules. How would you get an initial set of rules to work on?

15-5. Rule R1 is redundant with respect to rule R2 if any conclusion proved by R1 can also be proved by R2 in the same circumstances.

(a) Explain how R1 can have a different left side than R2 even when it is redundant with respect to R2.

(b) Give a criterion for the redundancy of R1 with respect to R2 that does not require execution of the rules.

15-6. (R,A,E) Discuss the overall suitability of artificial intelligence techniques as the *major* component in the following.

(a) An automatic car that will accelerate and steer on highways without human assistance.

(b) A generator of new cooking recipes.

(c) A "seduction advisor" expert system that advises you how to succeed with members of the opposite sex.

(d) A system to coordinate defense from nuclear missiles.

15-7. (H,P,G) Write a program to automatically build ("learn") a rule-based system from examples and near misses (nonexamples) of left-side predicates. Consider only predicate expressions without variables. Use Figure 15-3 as a guide, and break your problem into the following components:

> (1) a "comparer" that takes two inputs, a set of rules (a list of lists) and a description of some situation that is either an example or a near miss (a list of facts). It must compute the differences between the situation and the closest rule, the rule that matches the most facts of the situation. For that rule, it should list differences of four kinds: (a) additions (things that must be added to the rule to match a fact of the situation), (b) deletions (things that must be removed from the rule because they have no counterpart in the situation), (c) bindings (variables in a rule that can be matched to a constant argument of the same-named predicate in the situation), and (d) substitutions (constant arguments in a rule that can be matched to a different constant argument of the same-named predicate in the situation).
>
> (2) An "example handler" that takes the same two inputs as the comparer. It hands these two arguments immediately to the comparer, obtaining the list of differences of the situation from one particular rule. It then modifies the rule appropriately, or creates a new rule, to cover the situation. This done, the new generalized rule may cover anything handled by another rule, and the other rule can be eliminated. The example handler then returns the new set of rules as its result, together with a "gripe amount" indicating how happy it was with the rule modifications it made. If more than one modified set of rules seems equally reasonable, the example handler should permit backtracking into it.
>
> (3) A "near miss handler" analogous to the example handler. It should call the comparer, get back a set

of differences between a near miss situation and a rule, and modify (specialize) the rule appropriately so it won't apply to the situation. It should check that its changes don't cause contradictions among the rules. Like the example handler, it should return a new set of rules with a "gripe amount", and should give further answers (if any seem good) on backtracking.

(4) A "top-level searcher" that supplies examples and near misses to the rest of the system, searching over all possible permutations in the order of supplying them. There's a good cost function in the "gripe amounts" returned by the example and near miss handlers, and there's a good evaluation function in the number of rules plus the number of constants in rules. (Fewer rules means better generalization, and fewer constants mean more generality in rule application). This should use the A* program of Chapter 10, defining the successor function to call the example handler or near miss handler depending on whether the next situation chosen is marked as an example or a near miss.

Try learning rules for some simple diagnosis expert system.

Miscellaneous exercises covering the entire book

M-1. (A) Backtracking is important in some very different areas of artificial intelligence. For each of the following, say "yes" or "no" as to whether backtracking is usually necessary to implement them. (All these might use backtracking in Prolog implementations, but which need backtracking no matter what language they are implemented in?) By "backtracking" we mean returning to a processing state from which the current processing state is reachable.

(a) backward chaining

(b) forward chaining

(c) best-first search

(d) answering long queries efficiently by exploiting dependencies

(e) slot inheritance

(f) resolution among members of a set of clauses without variables

M-2. Lattices (directed graphs without cycles) are important in many different ways in artificial intelligence. They're important not only as a data structure inside the computer, but as a way of explaining what the computer is doing. Explain in 40 words or less what they are useful for in each of the following areas.

(a) search

(b) frames

(c) compiled rule-based systems, in a way different from uncompiled rule-based systems

(d) resolution theorem-proving

M-3. (R,A) Give the letter for the best match to each item in the first list from the choices in the second list. Each item should be matched to one and only one item. (Hint: doublecheck your answers by re-solving from bottom to top, and then by matching the second list to the first list.)

```
Horn clause
generate-and-test
constraint
forward chaining
default
depth-first control structure
intension
difference table
near miss
Skolem function
caching
dependency
multiple inheritance
decision lattice
Bayes's Rule
heuristic
hierarchical reasoning
agenda
```

```
a. a way of reusing results
b. a weak general-purpose rule or fact
c. contradictory reasoning is a danger
d. a theorem
e. example: prefer investigating further the last thing discovered
f. "or" with "not" in front of every expression but one
g. a fast implementation of rule-based systems
h. recommends an operator
i. example: the Prolog query ?- a(X), b(X), c(X).
j. used for existential quantifiers
k. lists candidates
l. a description of means-ends analysis
m. a predicate expression
n. valuable in debugging all artificial intelligence systems
o. reasoning about implications of facts
p. the abstract meaning of something
q. requires a stack
r. when two predicate expressions have some of the same variables
```

M-4. (R,A) Consider an expert system built to advise Americans which income tax forms and schedules to fill out. There are different ways satisfying the laws, and some are better for the taxpayer than others. (It matters how much a taxpayer earns and how much of their income is of a certain type. The taxpayer wants the possibility that costs the least money.) Using answers to some simple questions, an expert system could make recommendations without having to ask for a complete financial history. But its conclusions will be necessarily imperfect without such complete information.

(a) Which of the following techniques is most appropriate for this problem? Explain why.

1. and-or-not lattice
2. dependency-based backtracking
3. confusion-matrix conflict-resolution
4. breadth-first resolution strategy

(b) Which of the following techniques is *least* useful for this problem? Explain why not.

1. virtual facts
2. inference
3. depth-first control structure
4. concurrency

(c) Which of the following Prolog features is *least* useful in implementing an expert system in Prolog for this problem? Explain.

(a) **asserta** or **assertz**
(b) **repeat**
(c) **write**
(d) > [greater-than sign]

M-5. (A) Consider the problem of foot placement for a 6-legged robot. Suppose the robot has a vision system that studies the ground in terms of squares one square meter in area. For each such square the vision system (which you don't need to consider) decides whether that square is a good one to place a leg on. Suppose only 20 percent of the squares are suitable on the average, and suppose the robot can only extend its legs a certain distance forward. Suppose the vision system only examines squares within a rectangle centered in front of the robot, extending from the front of the robot to ten meters ahead, and ten meters off to the left, and ten meters off to the right. Consider the problem of how to use that information to plan a good sequence of foot placements to the centers of squares, keeping the robot moving ahead in an approximately straight line. The robot is not in a hurry but it doesn't want to ever back up.

(a) Explain which of these search strategies is best for this problem and why.

1. best-first search
2. weight-first search
3. breadth-first search
4. branch-and-bound search

(b) Which of the following are also helpful in step planning? Explain why.

1. means-ends analysis

2. caching
3. both generate-and-test and caching

(c) Which of the following Prolog features is *least* useful for implementing this step-planning (not the vision system) in Prolog? Explain why.

1. **is**
2. **assertz** (as opposed to **asserta**)
3. **cut** (!)
4. **repeat**

M-6. Many artificial intelligence applications require representation of causation. Suppose we use the Prolog predicate **causes(X,Y)**, where **X** and **Y** are names of events such that event **X** causes event **Y** to happen. **X** is assumed to be sufficient for **Y** to happen (so that any time **X** happens **Y** must necessarily happen too), but **X** is not necessary for **Y** to happen (**Y** may have other causes).

(a) Is **causes** transitive? Explain.

(b) Does **causes** inherit with respect to **part_of**? If so, in which direction? Explain.

(c) Consider an alternative representation of causation in Prolog as a rule with the "**:-**" symbol. For example:

```
event(lights_come_on) :- event(turn_wall_switch_to_on).
```

or better yet:

```
event(lights_come_on) :- event(turn_wall_switch_to_on),
 not(bulbs_burned_out), not(wiring_defective), not(switch_broken).
```

But give a major disadvantage of representing causation this way as a rule instead of with **causes(X,Y)** facts. (Give a "deep" problem; don't consider efficiency.)

M-7. (E) The conjunctions "and" and "or" are very important in artificial intelligence. Consider the conjunction "but".

(a) Suppose we let experts building an expert system use the word "but" in writing rules. How should we translate it into something a programming language like Prolog can handle? Be sure your answer covers most of the usual meaning of "but".

(b) Vocabulary variety makes computer output more comfortable for humans. Discuss when a computer could use "but" in natural language output from a program. In particular, consider a query-answering program. Compare with your answer to part (a).

(c) Would a "but" feature, in addition to "and" and "or", be useful in a logic programming language like Prolog?

M-8. (E) Discuss the possible application of artificial intelligence techniques to the following two problems:

(a) You are in a submarine and have various sensors. You want to guess the identity of all moving objects up to the limits of your sensor range.
(b) You are positioning army units for a possible battle. You have various kinds of personnel, and various kinds of weapons. You want to position units and equipment in a way that gives you the most tactical advantage.

For each problem, discuss which of the following specific techniques apply:

> automatic backtracking, generate-and-test, inheritance, defaults, rule-based systems, backward chaining, forward chaining, fixed-order conflict resolution, focus-of-attention conflict resolution, probabilities, depth-first search, breadth-first search, best-first search, A* search, means-ends analysis, frames, relaxation, dependency-based backtracking, resolution

M-9. (H,E) Consider the problem of building an automatic "commercial censor" system that will determine when a television is receiving a commercial, and will turn off the audio and video until the commercial is finished. (This would be nice when videotaping.) 95% accuracy within two seconds after a commercial starts or ends is desired. In addition to the frequency analyzer and the low-level vision processor mentioned, assume you have a fast computer with one million bytes of random-access memory and ten million bytes of disk storage. Conduct experiments (involving observation of at least 200 commercials) with a real television to help answer the following questions. An adequate answer to this problem will be 2000 words total.

(a) Categorize clues available from processing of the audio signal. Assume frequency-spectrum calculation hardware (technically, a discrete Fourier transform processor) is available.

(b) Categorize clues available from processing of the video signal. Assume a low-level vision processor (that groups homogeneous two-dimensional regions of a single picture, and calculates their average brightness and color, but slowly).

(c) Categorize clues available from correlation of the results of parts (a) and (b) with an accurate clock connected to the system.

(d) Describe a good control structure (or control structures) for an artificial intelligence system to do this job. Be sure to include whether backward, forward, or hybrid chaining is desirable; and whether depth-first, breadth-first, best-first, or A* searching is desirable.

(e) Suggest good uncertainty combination methods for such an artificial intelligence system. Are numbers a good idea? In particular, suppose each clue from parts (a), (b), and (c) is summarized by a single probability: how would you combine these three?

(f) Where might learning techniques help to automatically improve system performance with time? In particular, discuss how learning from examples can be done. Describe exactly what is learned: how it would be represented in the computer, how it would be modified upon new data, and so on. Remember, artificial intelligence methods are mostly nonnumeric, so your learning must be mostly nonnumeric.

(g) Describe how you would fairly validate the 95% accuracy claim. When would you run the system, how long, and how many times? Run some simple (not necessarily thorough) tests on a real television of some of the ideas you've developed previously. Explain how you could also use these testing procedures in debugging.

Some hints: often students are too superficial in their answers to essay questions like these, so err on the side of writing too much rather than writing too little. Don't give definitions of terms and other things that are easy to look up.

Go to book index

Appendix A: Basics of logic

Logic is the study of things that are either true or false and never anything in between. Such things are called *propositions* or *statements*. Propositions can be combined to form more complex propositions using "and", "or", and "not". The "and" of some propositions is true if each is true; the "or" of some propositions is true if one or more are true; and the "not" of a proposition is true if the proposition is false. Otherwise, the result is false. Figure A-1 summarizes these three operations for simple cases; together they define the *propositional calculus*.

Other operations can be built from these three. For instance, logical implication: A implies B if whenever A is true, B is true too. It's usually interpreted to mean (see discussion in Section 14.2) that this can only be false when A is true and B is false, so an equivalent proposition is "B or not A".

The "and", "or", and "not" operations obey some of the same laws as arithmetic operations. Order in an "and" or "or" doesn't matter, and you can do the operations in pieces, so "and" and "or" are commutative and associative. "And" is distributive with respect to "or", and vice versa:

> A and (B or C) equals (A and B) or (A and C)
> A or (B and C) equals (A or B) and (A or C)

This means that we can always "factor out" repeated propositions, or "factor in" a single proposition, and have an equivalent form. You can do this more than once for complex proposition, as for instance

> (A and B) or (C and D) equals ((A and B) or C) and ((A and B) or D)
> equals (A or C) and (B or C) and (A or D) and (B or D)

A *literal* is either a simple proposition (no "and"s, "or"s, or "not"s in its description) or the "not" of a single simple proposition. Then by repeated application of the distributive law, you can always convert any complex proposition to an "or", each item of which is an "and" of literals; this is *disjunctive normal form*, and it looks like this:

> (A and B and...) or (C and D and ...) or (E and F and ...) or ...

By repeated application of distributivity in just the opposite way, you can always get an "and" each item of which is an "or" of literals. This is *conjunctive normal form*, and it looks like this:

> (A or B or ...) and (C or D or ...) and (E or F or ...) and ...

where A, B, C, etc. are literals. Conjunctive normal form is easier to use in Prolog, and is especially important for the resolution methods in Chapter 14 of this book. In putting expressions into normal form, these simplifications help:

A and A equals A
A or A equals A
A and (not A) equals false
A or (not A) equals true
A and true equals A
A and false equals false
A or true equals true
A or false equals A
(not true) equals false
(not false) equals true

Here are four important examples:

A and (not(A) or B) equals (A and not(A)) or (A and B) equals A and B
A and (A or B) equals (A or false) and (A or B) equals A or (false and B) equals A
A or (not(A) and B) equals (A or not(A)) and (A or B) equals A or B
A or (A and B) equals (A and true) or (A and B) equals A and (true or B) equals A

Distributivity does not hold for "not"s with respect to "and"s and "or"s. But two related formulas hold (DeMorgan's Laws):

not(A and B) equals (not A) or (not B)
not(A or B) equals (not A) and (not B)

And

not(not(A)) equals A

hence two important relationships between "and" and "or":

not(not A and not B) equals not(not A) or not(not B) equals A or B
not(not A or not B) equals not(not A) and not(not B) equals A and B

"And"s, "or"s, and "not"s can be represented graphically with the standard gate symbols used in hardware design (see Figure A-2). Such a *logic gate* display can help you see patterns you wouldn't see otherwise. The left sides of the gates in the figure represent inputs (operands), and the right sides outputs (operation results). In such designs, it is usually assumed that lines crossing do not connect unless a there is a black dot at the crossing. Venn diagrams are another useful graphic representation; see Section 8.3.

So far we have only *propositional logic* or the *propositional calculus*. If we let propositions have variables, we get a more powerful kind of logic called *predicate logic* or the *predicate calculus*. We extensively illustrate predicate logic by example in this book, so we won't say much here, but it is important to explain the key feature of *quantification*. If some predicate expression is true for any value of an included variable X, then that the expression is *universally quantified* with respect to X. Another way of saying it is that the expression holds for every X. If an expression is true for some (but not necessarily all) values of an included variable X, then it is *existentially quantified* with respect to X. Alternatively we can say that there exists an X such that the expression is true.

Universal quantification has more practical applications than existential quantification, so we will refer to it more. Some additional laws of logic cover universal and existential quantification, but only two are particularly important in this book:

> the "not" of something existentially quantified is equivalent to the universal quantification of the negation of that something;
>
> the "not" of something universally quantified is equivalent to the existential quantification of the negation of that something;

As an example, suppose an appliance works properly. Then you can say that there does not exist a component X of the appliance such that X is faulty. But that is equivalent to saying that for every component X of the appliance, that component is not faulty.

Go to book index

Appendix B: Basics of recursion

Recursion is a valuable computer programming technique. It's a way to get a computer to do something over and over again. Recursion is tricky to understand at first, but once you get comfortable with it, it's easier and safer to program with recursion than with iterative techniques. We especially emphasize recursion in this book because Prolog has a natural affinity for it.

Recursion is a named piece of a program referring to itself. Why do this? Typically, to describe a task or job in the simplest possible way. A good way to explain a complicated job is to break it into components, and that's simplest when those components are the same sort of job. For this to work, we need to guarantee two things: first, that the new jobs are simpler than the old (for otherwise we wouldn't progress); and second, that by decomposition we'll eventually reach "easy" jobs that we can solve quickly without further decomposition. The easy jobs are called the *basis* of a recursion, and the decomposition into similar subjobs is called the *induction* of a recursion. (Recursion is closely related to a mathematical proof technique called *mathematical induction*, and that's where the terms come from.)

Here's an example (see Figure B-1). Suppose we have employee salaries in our computer, and we want to add 1000 to every one of them. Consider the easiest possible case (basis case) first. That's when there's only one salary, so we just add 1000 to it and store it. But suppose we have 732 salaries. One way to decompose the job (shown in the top diagram in Figure B-1) would be to add 1000 to the first salary, then establish a new job to add 1000 to the other 731 numbers. That new job isn't much simpler, but 731 *is* less than 732 so it is simpler. There are other ways to decompose the job too. For instance (as shown in the bottom diagram in Figure B-1), we can divide the job into equal halves: subjobs of adding 1000 to the first 366 items and adding 1000 to the last 366 items. Each subjob is the same sort of job as the original. Note that successful recursion just requires that the new jobs be smaller than the original job; it doesn't matter how many new jobs there are.

A recursion can have more than one basis step. For the previous example, a second basis step could handle two-item lists, so we wouldn't have to always decompose jobs to single-item lists eventually. In Prolog, basis conditions can be either facts (see Chapter 2) or rules (see Chapter 4), but induction conditions are always rules. Note that the first of the two decomposition methods in the last paragraph (binary decomposition in which one part is size 1) is the most common kind of recursive decomposition, because it's easy to implement despite the apparent inefficiency of its extreme uneveness of decomposition.

If you are having trouble understanding recursion, you should study some simple recursive programs and try to apply their form to new situations. The simplest form of recursion is *tail recursion*, in which there is only one induction (recursion) case, an extreme uneven decomposition, and its recursive call occurs last in the program. The first of the two ways to update 732 salaries is a natural application for tail recursion. Tail recursion is the commonest form of recursion, and has the advantage that it is easy to convert to iteration: the recursive call is like a branch or "goto" at the end of a loop. Recursive programs tend to be a little slower than their equivalent iterative counterparts, so that is one reason to convert them (though the recursive versions tend to be easier to understand). However, any recursion with more than one recursive (self-) reference is hard to convert to iteration, like the second (even-split) way of updating 732 salaries, or the means-ends analysis program in Chapter 11.

Recursion can define functions (in the mathematical sense) as well as subroutines. For instance, suppose the job is to compute the sum of 732 employee salaries (see Figure B-2). The basis case is when there is only one number, and the sum is that number. The induction case is when there are N numbers, N > 1; let's use tail recursion for it. That means we recursively decompose this problem into a job to find the sum of all but the first of numbers, and add to first number this total. So each level of recursion returns a value to the next highest level.

Chapters 4 and 5 introduce recursive programs in Prolog. One word of advice: students often have trouble with recursion because they worry too much about what the program is does during execution. You can usually follow the first recursive call, and maybe the second, but after that it gets confusing because it's hard to see where to "go back to" when one recursive call concludes. Human brains just aren't as well equipped as computers for this sort of reasoning. So stick to the *declarative* or *logical-truth* description of recursion used in this Appendix. Then if your recursive program doesn't work right, use the following checklist:

1. Do the cases meant to be handled by basis conditions work right?

2. Do the basis conditions cover *all* the easy cases that might occur through recursive decomposition?

3. Does the recursive decomposition really decompose the problem into *simpler* subproblems? (How many doesn't matter.)

4. If the recursive program has arguments, are the arguments in the recursive calls correctly different from the arguments at the top of the program?

Usually one of these things will be at fault if a recursive program is not working.

Go to book index

Appendix C: Basics of data structures

Data structures are ways of storing things inside a computer. The efficiency of an artificial intelligence program can be affected a good deal by the choices made for its data structures. We summarize the main kinds in Figure C-1. The many books on this subject provide further information.

Basic data structures are ways of storing small single items of data. Programming languages typically provide as basic data structures (or *data types*) integers, real numbers (in floating point notation), characters(alphanumeric), and bits (things that can be either true or false). Some programming languages also have pointers (actual memory addresses).

The simplest of composite data structures is the *array*. It's just a group of data items stored in successive locations in computer memory. The data items can be any of the aforementioned basic types. So we can speak of an *integer array*, a *real array* or a *character array*; the last is usually called a *string*. To find the data item in the Kth position in a plain (*one-dimensional*) array, you give the *index* K. For a *two-dimensional* array, you give two numbers, a *row* and *column* for a data item. Use of an array requires the start of the array and the length of each dimension.

But arrays are a poor data structure when you must frequently change the data. The *linked lists*, often called just *lists*, are better. Linked lists store sequences of data items like arrays do, but the items do not have to be in successive locations in computer memory; each is paired with a pointer giving the memory location of the next data item in sequence, with the last data item in the list having a special pointer called *nil*. So a list can be scattered around in computer memory, provided you keep item-pointer pairs together and store the location of the first data item. Linked lists need more space to store data items than an array would, but allow more flexibility in the use of computer memory. Lists are more common in artificial-intelligence programming than arrays, and in fact the programming language Lisp is built entirely around lists: lists even represent programs in Lisp.

Several common data structures are special cases of arrays and lists. If you only insert and delete items at the very end of an array or list, then you have a *stack* data structure. Stacks are essential for managing procedure calls and recursion in programming languages, though their operation is usually hidden from the user; such stacks contain pointers to memory locations of program pieces. If you only delete items at the very front of an array or list, and only insert items at the very end, then you have a *queue* data structure. Queues are used when you want to be "fair" to data items and want the data item that has been in the array or list the longest to be the first removed. So queues are first in, first out. Analogously stacks are last in, first out.

Arrays and lists order their items. An unordered set can be stored as an array or list in some random order, but there is an alternative. If data items can take on one of N data values--as for instance, data items for ages of employees in years, which must be positive integers less than 100--then we can represent sets as an N-element *bit array* (an array whose data items are bits) where a "true" in location K of the array means that the Kth possible is present in the set. Bit arrays can make set operations like intersection run very fast, and many computers have bit-handling instructions to speed things even more.

Data structures can arrange data in more complicated ways too. For instance, consider as data items the names of procedures, and suppose we want to represent how those procedures call one another in a nonrecursive program in which every procedure is called only once (e.g. Figure 7-3). A procedure can call several others, and each of those can call several others, and so on. This sort of branching data structure is called a *tree*. *Binary trees* are those for which every data item has one branch entering and either two or zero branches leaving; they're the most common kind of tree. A tree is usually stored in computer memory as an array of subarrays, each subarray holding a data value and pointers to records of other data items that are linked from it. Trees are useful whenever there are hierarchies or taxonomies in the data. Trees can also be built as an index to sequential data, when you want to be able to find things fast (*search trees*). Trees can be implemented in a list-processing language like Lisp by embedding lists within lists; the same thing can be done in Prolog.

If the branches of a tree can join together, then technically you don't have a tree but a *lattice*. Lattices and trees can be implemented the same way, and they are often confused. Lattices occur often in computer science. For instance, if the same procedure is used in several different places in a program, the hierarchy of procedures can be represented as a lattice. In general, lattices arise whenever you have partial information about ordering of things, but not complete ordering information.

If your data structure has loops so that you can return to some data item by following a sequence of pointers from it, then technically you have a *directed graph*, often just called a *graph*. Graphs are good at modeling arbitrary connections between things. You can implement graphs like trees, as arrays of subarrays containing pointers to other records. Another way is to extend a bit array to a *bit matrix*: for N possible data items, create on N by N matrix of bits so that the entry in row I and column J of this matrix is true if and only if there is a connection from data entry in row I to data item J in the graph. Some graphs require that you store data for each connection, not just for each data item, and for these graphs you can use a matrix but store data other than bits in it. Matrix array implementations are good whenever there are many connections between relatively few data items.

Still more complex data structures can be made by combining the aforementioned ones in various ways. Data items can themselves be data structures; for instance, data items representing English words can be character arrays. Lists can be items of lists themselves, and such *embedded lists* are used extensively in the language Lisp. If you want to represent a graph in Lisp, you can use a list of lists where each sublist contains a data item value followed by a list of pointers. And all the standard list processing of Lisp can be done analogously in Prolog. (But a word of warning: Lisp uses parentheses to denote lists while Prolog uses brackets.)

Recursion (see Appendix B) is useful for manipulating all these data structures. Lists and arrays suggest tail recursion in the form of a single basis case followed by a single induction case, where the basis case concerns a single-item array or list, and the induction case decomposes the problem by eliminating a single item from the list or array. Trees, lattices, graphs, and embedded data structures require more complicated forms of recursion involving either multiple induction cases or multiple recursive procedures. Data structures books provide classic recursive algorithms for all these things and they should be consulted for most simple needs instead of your trying to write programs yourself.

Go to book index

Appendix D: Summary of the Prolog dialect used in this book

The Prolog examples in this book use a subset of the "core Prolog" or "standard Prolog" originally developed in several implementations at Edinburgh University and covered in W.F. Clocksin and C.S. Mellish, *Programming in Prolog*, second edition, Springer-Verlag, 1984. Substantially similar dialects are provided for many computers, and include Quintus Prolog, Logicware MPROLOG, DEC-10 Prolog, and C-Prolog. (Programs in this book were tested in Edinburgh C-Prolog on a DEC VAX-780.) Borland International Turbo Prolog is close to this standard, but doesn't support treating rules as data (Section D.8) and has some other differences. Micro-Prolog has major differences; see Appendix E. For compatibility with these dialects and easy comprehension by the reader, the Prolog subset of this book omits many common and useful features, including other arithmetic operations, input and output, structures (as opposed to embedded lists), functors, modules, and many debugging tools; see the Prolog books in Appendix F for information about these.

This Appendix is summarized in Figure D-1.

D.1 Managing facts and rules

We assume Prolog programs (consisting of rules and facts) are written outside of the Prolog interpreter. You use your favorite editor to create a text file containing them. Then you start up the Prolog interpreter (how to do this is implementation-dependent) and load in the files you want by querying **consult**, whose one argument is the name of the editor-created file you want to load. If you have several files to load, do **consult** several times.

To list all the facts and rules in the Prolog database, query **listing** with no arguments. To list only the facts and rules with a particular predicate name, query with that name.

To add a fact or rule to the Prolog database from within the interpreter, query **asserta** or **assertz**. Built-in predicate **asserta** adds the new fact at the front of the database, and **assertz** adds it to the end. To remove a fact from the Prolog database, query **retract** with that fact an argument. To remove all the facts and rules with a particular predicate name, query **abolish**, whose first argument is the predicate name and whose second is the number of arguments the first-argument predicate has. All four of these predicates fail on backtracking, so the only way to undo an **asserta** or **assertz** is a **retract**.

D.2 The format of facts, rules and queries

Facts, rules, and queries are all built of the same thing: predicate expressions. An expression consists of a predicate name (starting with a lower-case letter) optionally followed by one or more arguments separated by commas and enclosed in parentheses. Arguments that are constants must be numbers or start with a lower-case letter, and arguments that are variables must start with a capital letter. Facts are just predicate expressions followed by a period ("**.**").

Queries are things you type after the Prolog interpreter types "**?-** " on your terminal; query mode is the interpreter's normal mode. A query is a single predicate expression, or several expressions linked with commas or semicolons, concluded by a period and carriage return. Commas mean "and" and semicolons mean "or". You can also specify that the opposite of something be true with the **not** predicate of one argument; its argument is the predicate expression you want to fail | REFERENCE 1|. .FS | REFERENCE 1| The **not** is denoted with prefix "/+" in Quintus Prolog, and it won't work with inside unbound variables in Turbo Prolog. .FE

If you type a semicolon after the interpreter types an answer to a query, the interpreter will go back to work and try to find another answer to the query. You can keep typing semicolons as long as there are more answers. If the interpreter can't find an answer, it will type **no**.

Rules consist of a predicate expression followed by a space, the symbol "**:-**", another space, a query, and a period. The stuff before the **:-** is the *left side* and the stuff after is the *right side*. Rules can be read as "if the right side is true then the left side is true".

D.3. Program layout

Spaces and carriage returns can be inserted at many places in Prolog programs without changing the meaning. However, it's conventional not to put them in normal predicate expressions (except within strings). A space is generally put after every "and"-comma and "or"-semicolon in a query or rule, as well as before and after the rule symbol **:-** and the arithmetic-assignment **is**.

Another convention of Prolog programming is to leave blank lines in programs between rules and facts beginning with different predicate names; this makes it easier to see which lines go together.

Comments are preceded with a "**/***" and followed by a "***/**".

D.4. Lists

Arguments to predicate expressions can be lists. Lists are indicated by the brackets "**[**" and "**]**"; commas (not to be confused with the commas between predicate arguments) separate the list items. The list of no items "**[]**" is called the empty list. Lists of any length can be represented by variables, or they can be referred to by the notation **[<front_items>|<sublist>]**. Here **<front_items>** represents an item, or several items separated by commas, at the front (left end) of a list, and **<sublist>** is a variable representing the arbitrarily-long rest of the list. Items within a list can be variables too.

D.5. Numbers

Arguments to predicate expressions can be numbers. (Some dialects accept only integers, and some others only accept decimals with digits before and after the decimal point.) Prolog handles numbers with infix notation, to make things easier to read. (Predicate expressions are prefix notation; that is, the predicate name comes first.) The infix comparison predicates are **=** (equals), **>** (greater than), **<** (less than), **=<** (less than or equal to), and **>=** (greater than or equal to). Any variables used in these comparisons must be bound to values.

Arithmetic assignment is handled by the **is** infix symbol | REFERENCE 2|. .FS | REFERENCE 2| Not available in Turbo Prolog: **=** is dual-purpose. .FE To the left of the **is** and separated by a space must be the name of a variable; to the right and separated by a space must be an arithmetic expression of constants and/or bound variables. The arithmetic expression is evaluated, and its value bound to the left-side variable. The arithmetic expression must be in infix form, using the standard infix arithmetic symbols **+**,**-**, *, and **/**; parentheses can group terms.

D.6. Output and input

To write something on the terminal screen use the **write** predicate of one argument. The argument can be a word, list, number, string, bound variable (for which it prints the current binding), or even unbound variable (for which it prints the current internal code of the variable name). To get a carriage return on the terminal, query the **nl** (that stands for "new line") predicate of no arguments. The predicate **read** of one argument, a variable, reads in what the user types on the terminal keyboard and binds the variable to it. What the user types must end with a period, just like everything else in Prolog.

D.7. Strings

Character strings are denoted by single-quotation marks or apostrophes ("'") around some characters and punctuation marks. Strings can contain capital letters, commas, periods, and other punctuation marks in any order, so they're less restricted than predicate names, variable names, and constants.

D.8. Treating rules and facts as data | REFERENCE 3|

.FS | REFERENCE 3| Not supported in Turbo Prolog. .FE The predicate **clause** matches its first argument to the left side and its second argument to the right side of a rule or fact in the database (if a fact, its second argument is set to the empty query which always succeeds, "true") | REFERENCE 4|. .FS | REFERENCE 4| Some Prolog dialects won't accept a **clause** expression unless the left-side predicate name is filled in. But you can get the same effect by iterating over the list of all predicate names, trying each one for the left side. .FE The infix "univ" operation represented by the symbol "**=..**" converts queries into lists and vice versa; the left side of this symbol is the query, and the right side the list. If the left side is bound it is converted into its equivalent list, and if the right side is bound it is converted into its equivalent query. Related to this, the **call** predicate of one argument takes a query and executes it as if it were typed on the terminal.

D.9. Miscellaneous predicates

The **fail** predicate of no arguments provides a predicate expression that always fails, forcing backtracking. The "true" predicate always succeeds. The cut predicate (symbolized by "!") of no arguments forces throwing away of certain backtracking information; see Section 10.7 for a detailed discussion. The **var** predicate of one argument succeeds if its argument is unbound. The **number** predicate of one argument succeeds if its argument is a number. The **halt** predicate of no arguments stops the Prolog interpreter and returns you to the operating system; it's the recommended way to finish a session.

D.10. Definable predicates

Besides the preceding built-in predicates, some predicates defined in this book in terms of others are built-in in many Prolog dialects. The **forall** predicate of two arguments (Section 7.12) checks whether universal quantification holds for some predicate expression; specifically, it checks whether every set of variable bindings that satisfies its first-argument predicate also satisfies its second-argument predicate expression. The **doall** predicate of one argument (Section 7.12) repeatedly calls its argument predicate expression until that argument fails. The **repeat** predicate (Section 10.8) of no arguments always succeeds, and always succeeds afresh on backtracking. The **bagof** predicate | REFERENCE 5| .FS | REFERENCE 5| Called **findall** in Clocksin and Mellish. .FE of three arguments (Section 10.6) collects into a list all values to which a variable can be bound to satisfy a predicate expression. The first argument is the variable, the second is the expression (somewhere inside which must be the variable), and the third argument is the list of all possible values, the result of the computation.

D.11. Debugging

Since the activities of Prolog interpreters can be complicated, good debugging facilities are essential. These facilities differ considerably between dialects. But **trace**, **spy**, and **leash** features are common. These are built-in Prolog predicates that are queried to cause certain side effects. Predicate **trace** is a predicate of no arguments that makes the Prolog interpreter print a running account of everything it is doing; **notrace** undoes it. Predicate **spy** is a predicate of one argument, a predicate name, that makes the Prolog interpreter print everything happening with that predicate; **nospy** with that same argument undoes it. To spy on several predicates, do several **spy** commands. The **trace** and **spy** printouts report calls, successes, failures, and backtracks involving predicate expressions with a given predicate name. The **leash(on)** predicate expression tells the interpreter to stop and pause after every printout line generated by a **trace** or **spy**; **leash(off)** undoes it.

Go to book index

Appendix E: Using this book with Micro-Prolog

Micro-Prolog is a dialect of Prolog designed to run in a small amount of memory on small computers. It looks considerably different, but its execution (semantics) is very similar. Micro-Prolog is designed primarily for non-programmers, people who aren't computer experts. So it is less elegant than standard Prolog, and doesn't have some of the advanced features, but it's easier to learn to use.

In Micro-Prolog, querying is not the default mode, so you must indicate a query by a "?" on the front of the line. You must use parentheses to group things, a lot like the language Lisp. You must surround the query with parentheses. The predicate name goes inside the parentheses used in standard Prolog to group arguments, and spaces are used instead of commas to separate the arguments and the predicate name from the first argument. So the standard Prolog query

```
?- a, b(X), c(X,Y).
```

becomes in Micro-Prolog

```
? ((a) (b X) (c X Y))
```

Note a period is not used.

The symbol table is limited in Micro-Prolog, so variable names can only be X, Y, Z, X1, Y1, Z1, X2, Y2, Z2, and so on. Micro-Prolog will not print out variable bindings unless you force it with the equivalent of the standard-Prolog **write**, the "P" predicate. So to print out X and Y for the preceding query you must say

```
? ((a) (b X) (c X Y) (P X) (P Y))
```

Predicate names in Micro-Prolog can be capitalized or uncapitalized, just as long as they aren't a possible variable name. (Built-in predicates are usually all capitals.)

Rules in Micro-Prolog look just like queries without the question mark. But the first predicate is interpreted as the left side ("then" part), and the other predicates as the right side ("if" part). So the standard-Prolog rule

```
a(X) :- b(X), c(X,3), d(Y,X).
```

becomes in Micro-Prolog

```
((a X) (b X) (c X 3) (d Y X))
```

Figure E-1 gives a table of approximate equivalents between standard Prolog and Micro-Prolog. Micro-Prolog manuals provide further information.

Go to book index

Appendix F: For further reading

We list a few books and articles recommended for additional reading on the topics of this book. We've grouped related entries together.

General artificial intelligence:

> Winston, P. H., *Artificial Intelligence*. second edition. Reading, Mass.: Addison-Wesley, 1984. Useful for basic understanding of artificial intelligence.
>
> Charniak, E. and McDermott, D., *Introduction to Artificial Intelligence*. Reading, Mass.: Addison-Wesley, 1985. A well-balanced, reasonably detailed treatment of the field at a more advanced level than ours. Also a good source for further information. Uses examples in Lisp.
>
> Gevarter, W. B., *Intelligent Machines: An Introductory Perspective of Artificial Intelligence* and Robotics. Englewood Cliffs, N. J.: Prentice-Hall, Inc., 1985. A short summary of the key ideas in artificial intelligence.
>
> Reitman, W., (ed.), *Artificial Intelligence Applications for Business*. Norwood, N. J.: Ablex, 1984. An interesting collection of articles surveying of the major commercial applications of artificial intelligence.
>
> Feigenbaum, E., and McCorduck, P., *The Fifth Generation: Artificial Intelligence and Japan's Computer Challenge* to the World. Reading, Mass.: Addison-Wesley, 1983. A non-technical, general-audience book arguing for the critical importance of artificial intelligence in the world's future. Controversial but interesting reading.

The language Prolog:

> Clocksin, W. and Mellish, C., *Programming in Prolog*. second edition. Berlin, B.R.D.: Springer-Verlag, 1984. A well-written introduction to Prolog as a programming language, including features not covered in our book. Uses a dialect containing ours.
>
> Sterling, L. and Shapiro, E., *The Art of Prolog*. Cambridge, Mass.: MIT Press, 1986. A more detailed introduction to Prolog.
>
> Bratko, I., *Prolog Programming for Artificial Intelligence*. Reading, Mass.: Addison-Wesley, 1986. Despite its title, mostly a programming language textbook in the manner of

Clocksin and Mellish. But it has additional chapters on implementing search in Prolog (Chapters 11, 12, 13, and 15) corresponding to much of our Chapters 9 and 10, a chapter (Chapter 14) on expert systems covering parts of our Chapters 7 and 8, and some material on resolution (Section 16.3) corresponding to the program in our Chapter 14.

Ennals, R., *Artificial Intelligence: Applications to Logical Reasoning and Historical* Research. New York: Wiley, 1985. A good non-threatening introduction to Prolog for students without strong computer science backgrounds. Helpful for people finding the early chapters of our book tough going.

Clark, K., and McCabe, F., *Micro-Prolog: Programming in Logic*. Englewood Cliffs, N. J.: Prentice-Hall International, 1984. An introduction to the Micro-Prolog dialect.

Cohen, J., "Describing Prolog by Its Interpretation and Compilation, *Communications of the ACM, 28,* 12 (December 1985), 1311-1324. A good survey of Prolog from a programmer's point of view.

Subareas of artificial intelligence:

Waterman, D., *A Guide to Expert Systems*. Reading, Mass.: Addison-Wesley, 1986. A good introduction to expert systems, though not very technical. Extensive bibliography.

Hayes-Roth, F., Lenat, D., and Waterman, D., (eds.), *Building Expert Systems*. Reading, Mass.: Addison-Wesley, 1983. A more detailed consideration of issues in building expert systems.

Winograd, T., *Language As a Cognitive Process, Volume I: Syntax*. Reading, Mass.: Addison-Wesley, 1983. Overview of the grammar (syntax) side of the natural language subarea.

Schank, R. and Abelson, R., *Scripts, Plans, Goals, and Understanding*. Hillsdale, N.J.: Lawrence Erlbaum, 1977. An integrated discussion of natural-language semantics, frames, and planning. Quite readable.

Special Section on Architectures for Knowledge-Based Systems, *Communications of the ACM, 28,* 9 (September 1985), 902-941. Contains three good overview articles on frames, rule-based systems, and logic programming.

Ballard, D. and Brown, C., *Computer Vision*. Englewood Cliffs, N.J.: Prentice-Hall, 1982. An overview of the vision subarea of artificial intelligence.

Wos, L., Overbeek, R., Lusk, E., and Boyle, J., *Automated Reasoning: Introduction and*

Applications. Englewood Cliffs, N. J., Prentice-Hall, 1984. Discusses use of a general-purpose theorem prover (based on extensions of resolution) for making inferences for interesting problems. A friendly book with lots of interesting examples, but its theorem-prover is more powerful than Prolog and so programming it is different.

Maier, D., "Databases in the Fifth Generation Project: Is Prolog a Database Language?", in *New Directions for Database Systems*, Ariav, G. and Clifford, J., eds. Norwood, N.J.: Ablex, 1986, 18-34. Good survey of the considerations necessary to make Prolog efficient with large databases of facts and rules.

Fishman, D., "The DBMS--Expert-System Connection", in *New Directions for Database Systems*, Ariav, G. and Clifford, J., eds. Norwood, N.J.: Ablex, 1986, 87-101. Explains how database design considerations are important for expert systems.

Michalski, R., Carbonell, J., and Mitchell, T., eds., *Machine Learning: An Artificial Intelligence Approach*. Palo Alto, Ca.: Tioga, 1983. Collection of articles summarizing the learning subarea.

Goldberg, A., and Robson, D., *Smalltalk-80: The Language and Its Implementation*. Reading, Mass.: Addison-Wesley, 1983. An overview of the object-oriented programming language Smalltalk.

Winston, P., and Horn, B., *Lisp*, second edition. Reading, Mass.: Addison-Wesley, 1984. A sequel to Winston's *Artificial Intelligence* which shows how to write some interesting artificial intelligence programs in Lisp. It illustrates well how Lisp programming is different from Prolog programming.

General background:

Mendelson, E., *Introduction to Mathematical Logic*. Princeton, N.J.: Van Nostrand, 1964. Just one of many good textbooks on logic from a mathematical perspective.

Tenenbaum, A., and Augenstein, M., *Data Structures Using Pascal*. Englewood Cliffs, N.J.: Prentice-Hall, 1981. Good coverage of the basics of data structures and recursion.

Zwass, V., *Introduction to Computer Science*. New York: Barnes and Noble, 1981. Basic survey of the fundamentals of computer science necessary for understanding artificial intelligence.

Wulf, W., Shaw, M., Hilfinger, P., and Flon, L., *Fundamental Structures of Computer Science*. Reading, Mass.: Addison-Wesley, 1981. A more advanced presentation than the preceding. Good coverage of data structures, recursion, and implementation of traditional programming languages.

Kolman, B. and Busby, R., *Introductory Discrete Structures with Applications*. Englewood Cliffs, N.J.: Prentice-Hall, 1987. Survey of the mathematical fundamentals necessary for artificial intelligence.

Longley-Cook, L., *Statistical Problems and How To Solve Them*. New York: Barnes and Noble, 1970. Just one of many good introductions to statistics and probability. This one is quite friendly.

Lindsay, P. and Norman, D., *Human Information Processing*. New York: Academic Press, 1972. Well-written introduction to the tie-ins between artificial intelligence and research in human cognitive psychology.

Go to book index

Appendix G: Answers to selected exercises

(Exercises with answers were marked with an "A".)

2-1. The **color** fact is better, because colors are less based on judgment and context than sizes are. We can define colors by frequency ranges of light waves, but there are no absolute standards of size: a "big" ship is a lot bigger than a "big" molecule. So if you want to refer to sizes in an artificial intelligence program, you probably should use something more precise.

2-2. Answer (ii). For format (i) the predicate names are just as general as the argument names, meaning many separate predicate names are necessary. And there's no internal indication with format (i) that these are supposed to be memo facts. But format (iii) is too general--any fact can be written this way--and the predicate "fact" isn't informative. Also, (ii) is typical of the predicates used in the chapter.

2-5. The speaker is forgetting that arithmetic operations are functions, and the function result must be an explicit argument in predicate expressions. So one-argument operations become two-argument predicate expressions, and two-argument operations become three-argument expressions. Other than that, the speaker's argument is pretty much correct: logic is part of mathematics, and Prolog is closely related to logic.

2-6. Write each **ship** fact as five two-argument subfacts, each with first argument the original first argument of the **ship** fact. (That is, use the ship name as a *key*.) The second argument will be:

--ship location for predicate name **location**;
--time at location for predicate name **time**;
--date at location for predicate name **date**;
--ship color for predicate name **color**;
--ship commander for predicate name **commander**.

So for instance

```
ship(enterprise,15n35e,1200,160285,gray,j_kirk).
```

becomes

```
location(enterprise,15n35e).
time(enterprise,1200).
date(enterprise,160285).
color(enterprise,gray).
commander(enterprise,j_kirk).
```

2-7. You might want to reason about a hypothetical situation, a situation not currently true. Advance reasoning might help you plan what to do if the situation occurs, saving you time then and letting you figure the best thing to do. Reasoning about hypothetical situations is also useful for debugging programs. For such reasoning to work, the set of hypothetical facts must be consistent: they can't be false in the sense of being self-contradictory.

3-1. (a) Use DeMorgan's Law:

```
?- not(a), not(b).
```

(b) Use DeMorgan's Law and rearrange:

```
?- (a;b), (not(a);not(b)).
```

3-2.(a)

--What job bosses the boss of a CS professor?
--What person is Rowe's boss?
--Who is the incumbent dean of either IP or IPS? (querier probably doesn't know which is the proper abbreviation)
--What person either has the provost or the IPS dean as a boss?
--What job other than Shrady's is bossed by the superintendent?

(b) First answers found, in order:

```
X=cschairman
Y=dean_ips

X=csprofessor
Y=cschairman
Z=lum

X=marshall

J=cschairman
P=lum

P=director_milops
```

3-5. (a) NM (every possible pair).

(b) N (N-1 for each new choice of **X**, and then one for the final query failure).

(c) NM (the two parts of the query are independent).

(d) The minimum of M and N.

(e) N--it's just like part (b).

(f) 0. The arguments to **p** facts may be totally different from the arguments to **q** facts.

3-6. (a) "Joe" is a constant in the queries, and "someone" is a variable. Every time a new Prolog query is asked, the variable bindings from the previous query are thrown away, and so this only affects the "someone" query. The mistake or *fallacy*, is in assuming such bindings persist.

(b) Assuming that a student gets a single grade on a single test:

```
?- grade(P,1,G), not(grade(P,1,a)).
```

(c) No, this isn't correct. Different people may have taken the two tests: some students may have dropped the course and others may have added the course in between. So the meaning of "class" may have changed.

3-9.(a) **X=a, Y=d, Z=e**. The first three **r** facts all eventually fail when used to match **X** and **Y** in the first predicate expression in the query. **X=a** with **Y=b** fails in the third query expression, **X=a** with **Y=c** fails in the fourth query expression, and **X=b** with **Y=a** fails in the second query expression.

(b) Four times: for **Y=b** and **Z=c**, **Y=b** and **Z=d**, **Y=c** and **Z=d**, and **Y=c** and **Z=c**. Notice that even though the matching for the second predicate expression can't affect the success of the third expression, the Prolog interpreter will always backtrack there from the third because it's the immediately previous expression--that is, the interpreter always backtracks *chronologically*.

4-2. (a) Make the left side a fact.

(b) Delete the rule entirely--it never does any good.

4-5. No, not if **a** is queried with an unbound argument and there is some **c** fact in the database, say **c(3)**. Then the one-rule form binds **X** to 3 in **a**, but the two-rule form can't. But when **a** is queried with a bound argument, the two forms give the same answers.

4-6. (a) Two different meanings of the word "mayor" are being confused: the job description and the person currently occupying that office. These meanings can't be equated. We'll discuss this distinction more in Chapter 12.

(b) The phrase "a movie star" acts like a noun, whereas "a little stupid" acts like an adjective. So "Clint is a movie star" must be represented by an **a_kind_of** relationship predicate, whereas "John is a little stupid" must be represented by a property predicate.

4-7. (a) A definition with two things "and"ed together on the right side.

(b) Two different facts about a department chairman (notice they do not define "chairman", just give properties of one).

(c) Here the "and" is more like an "or". So you could write two rules with the same left side (representing "a way to put out a fire") and different right sides for the different methods. Alternatively but less directly, you could write two facts asserting two different "fire methods".

(d) Two **a_kind_of** facts.

(e) This is an unusual situation in English, an *idiom* in which the words don't mean what they literally say. Here Tom and Dick are not only friends, but implied friends of one another. So two interrelated facts are being asserted, or just a disguised relationship-predicate fact.

4-10.(a) upward only. If some of Set have Property, then those same items will be present in a set including Set.

(b) No basis conditions are needed because facts will provide them. The induction step looks like this:

```
some(Set,Property) :- contains(Set,Set2), some(Set2,Property).
```

(c) downward only. If all of Set have Property, then all the items of any subset of Set have the same property.

(d) neither way. "Most" is a statistical generalization about a set, and there's never any guarantee that a statistical generalization about one set applies to a related set. (A subset might just happen to contain atypical items.)

4-11. (a) Yes, something inside something else inside a third thing is inside the third thing.

(b) Yes, it in inherits downward in one sense. If A is in front of B, and B contains C, then A is in front of C. However, downward inheritance doesn't work the other way: if A is in front of B and B is inside C, then A is not necessarily in front of C because C could contain A too: let A=battery, B=motor, and C=entire car.

(c) To better talk to people (as when giving them advice) because people understand nonnumeric descriptions better. And if people were given coordinates to locate things, they would need to do a lot of measurements from reference points, which is bothersome. Furthermore, it's tricky to compute object relationships from Cartesian coordinates describing object shapes; a lot of mathematics is necessary, especially when it won't do to just store the center of irregularly shaped objects. So it would be awkward for the computer to store coordinates internally and convert them to physical relationships to talk to people.

On the other hand, Cartesian coordinates are more precise descriptors than relationship predicates: you can compute more accurate spatial relationships, ways to reach parts, etc. Cartesian coordinates might take less memory space than relationship information since you wouldn't have to represent information about an object more than once; for N objects in a car there are |N * (N - 1)| relationships between any two, though inheritance could cut that down. And there are lots of ways to compress numbers to make storing them even more efficient, like keeping relative coordinates of parts. It's not true that relationship predicates are more car-independent (that is, the same description could apply to many different cars) than Cartesian coordinates, since coordinates could have associated tolerance intervals to make them more fuzzy. But "user-friendliness" is the chief disadvantage of Cartesian coordinates.

4-13. (a) The point of this part of the problem is to note the varied forms the same predicate can take. Only two predicates are needed: the **part_of** of Section 2.6, and a **contains** predicate that says that some of an object is a particular material (if you wanted to be fancy, you could have an additional predicate indicating that *all* of an object is of a certain material, but that's not necessary for the questions in part (d)). Note the statements refer to an Acme hotplate, so somehow you should say this.

```
part_of(acme_hotplate_cord,acme_hotplate).
part_of(acme_hotplate_body,acme_hotplate).
part_of(acme_hotplate_heating_element,acme_hotplate_body).
part_of(acme_hotplate_cover,acme_hotplate_body).
part_of(acme_hotplate_knob,acme_hotplate_cover).
part_of(acme_hotplate_wire,acme_hotplate_cord).
part_of(acme_hotplate_insulater,acme_hotplate_cord).


contains(acme_hotplate_heating_element,metal).
contains(acme_hotplate_knob,plastic).
contains(acme_hotplate_wire,metal).
contains(acme_hotplate_insulater,fiber).
```

(b)

```
?- contains(X,metal).
X=acme_hotplate_heating_element;
X=acme_hotplate_wire;
no.
?- part_of(X,acme_hotplate_body).
X=acme_hotplate_heating_element;
X=acme_hotplate_cover;
no.
```

(c)

```
part(X,Y) :- part_of(X,Y).
part(X,Y) :- part_of(X,Z), part(Z,Y).


contains(X,M) :- part_of(Y,X), contains(Y,M).
```

If you use a predicate for things entirely composed of a particular material--call it **all_contains**--then you need one more rule:

```
contains(X,M) :- all_contains(X,M).
```

(d) The difficulty in the second question is that any **not** must refer to a bound variable. The apparent context is parts of the Acme hotplate. So first define:

```
thing(X) :- part_of(X,Y).
thing(X) :- not(part_of(X,Y)), part_of(Y,X).
```

That is, a "thing" is anything mentioned in **part_of** facts. Here are the results for all three questions:

```
?- contains(X,plastic).
X=acme_hotplate_knob;
X=acme_hotplate;
X=acme_hotplate_body;
X=acme_hotplate_cover;
no.

?- thing(X), not(contains(X,fiber)).
X=acme_hotplate_body;
X=acme_hotplate_heating_element;
X=acme_hotplate_cover;
X=acme_hotplate_knob;
X=acme_hotplate_wire;
no.

?- contains(X,metal), contains(X,fiber).
X=acme_hotplate;
X=acme_hotplate;
X=acme_hotplate_cord;
no.
```

5-1. Note no **is** or = is needed.

```
max(X,Y,Z,X) :- not(Y>X), not(Z>X).
max(X,Y,Z,Y) :- not(X>Y), not(Z>Y).
max(X,Y,Z,Z) :- not(X>Z), not(Y>Z).
```

5-2. Write as facts instead of variables. Easy fact representation is a big advantage of Prolog over languages like Pascal.

```
translate(1,integer_overflow).
translate(2,division_by_zero).
translate(3,unknown_identifier).
```

In general, **is** and = should rarely be used for anything other than arithmetic and an occasional local variable. Instead, use the automatic facilities of Prolog for checking and binding values.

5-8.(a) With rules for inferring both starts and ends from the other, we must be careful to avoid infinite loops. We can do this by defining new predicates **inferred_end** and **inferred_start**.

```
inferred_end(E,T) :- end(E,T).
inferred_end(E,T) :- start(E,Tstart), duration(E,Dur),
  T is Tstart + Dur.

inferred_start(E,T) :- start(E,T).
inferred_start(E,T) :- end(E,Tend), duration(E,Dur), T is Tend - Dur.
```

(b) The second event must be completely over before the first event starts or else **after** is not a proper term to use in English.

```
after(E1,E2) :- inferred_end(E2,E2end), inferred_start(E1,E1start),
  E1start > E2end.
```

(c) The usual English meaning of **during** is that an event occurred entirely inside the duration of the other.

```
during(E1,E2) :- inferred_start(E1,E1start), inferred_start(E2,E2start),
  inferred_end(E1,E1end), inferred_end(E2,E2end),
  E1start > E2start, E1end < E2end.
```

(d) The rules should come after all the facts, so facts will always be preferred to answer queries. The order of the rules among themselves doesn't matter except for the two **inferred_start** and **inferred_end** rules, because order of rules having the same predicate name is the only thing the Prolog interpreter considers--it indexes facts and rules with the same predicate name.

5-9.(a) **speed(<gear_number>,<speed_in_rpm>)**. Positive speeds can be clockwise, negative counterclockwise.

(b)

```
speed(G,S) :- same_shaft(G,G2), speed(G2,S).
```

(c)

```
speed(G,S) :- meshed(G,G2), teeth(G,TG), teeth(G2,TG2), speed(G2,S2),
  S is 0 - ( S2 * TG2 / TG ).
```

(d) Use the following database:

```
speed(g1,5000).


same_shaft(g2,g1).


meshed(g3,g2).
meshed(g4,g2).


teeth(g1,100).
teeth(g2,30).
teeth(g3,60).
teeth(g4,90).


speed(G,S) :- same_shaft(G,G2), speed(G2,S).
speed(G,S) :- meshed(G,G2), teeth(G,TG), teeth(G2,TG2), speed(G2,S2),
  S is 0 - ( S2 * TG2 / TG ).
```

and issue this query:

```
?- speed(g4,S).
```

The one **speed** fact doesn't apply, so the first **speed** rule is tried. But there is no **same_shaft** fact with **g4** as first argument, so the rule fails. Now the second **speed** rule is tried. Gear g2 can be matched to **g2**, with **TG=90** and **TG2=30**. Then we must find the speed of **G2=g2**. No fact applies, but we can use the first **speed** rule: **g2** is the first argument to a **same_shaft** fact having **g1** as second argument, so its speed is the speed of **g1**. And a fact says the speed of **g1** is 5000. So the speed of g2 is 5000, and we can do the calculation in the second "speed" rule as

```
S = 0 - ( 5000 * 30 / 90 ) = -1667 rpm
```

and the variable **S** in the original query is bound to that number.

(e) You could get infinite loops if you had extra redundant facts, like if you had both **meshed(g2,g1)** and **meshed(g1,g2)**. You could also get infinite loops if you had more than one inference rule of either the previous types, as for instance if you had

```
speed(G,S) :- same_shaft(G,G2), speed(G2,S).
speed(G,S) :- same_shaft(G2,G), speed(G2,S).
```

Generally speaking, gears are intended to transmit power in one particular direction, so you only need one rule. You can encode that direction in the **meshed** facts.

(f) The gears wouldn't turn or would self-destruct; such a contradiction of rotation speeds is an impossibility.

5-14. (a)

```
member(X,L) :- append(L2,[X|L3],L).
```

(b)

```
last(X,L) :- append(L2,[X],L).
```

(c)

```
deleteone(I,L,DL) :- append(L1,[I|L2],L), append(L1,L2,DL).
```

(d)

```
before(X,Y,L) :- append(L2,[X|L3],L), append(L4,[Y|L5],L3).
```

5-17. (a) It replaces every other word in a list with the word "censored", binding the result to the second argument. The rule (third line) just says if you can find an **X** and **Y** at the front of your list first argument, change the **Y** to the word "censored" after recursively doing the same on the rest of the list.

That's a declarative analysis. A procedural understanding comes from figuring out what happens with simple example lists:

```
?- mystery([],Z).
Z=[]
?- mystery([a],Z).
Z=[a]
?- mystery([a,b],Z).
Z=[a,censored]
```

For the last example, the third rule applies for the first time. So **X** is matched to **a**, **Y** is matched to **b**, and **L** is matched to **[]**, the empty list. The rule involves a recursive call with first argument **L**, but **L** is the empty list, it's the same situation as the first example preceding, and **M** is bound to the empty list too. So the rule says to take **M** and **X** and the word "censored" and assemble them into the result, the list whose first two items are **X** and "censored", and whose remainder is **M**. But **M** is empty, so the result is **[a,censored]**.

Now consider the next hardest case, an example list with three items:

```
?- mystery([a,b,c],Z).
```

The third rule must be again used. It matches **X** to **a**, **Y** to **b**, and **L** to **[c]**, the list containing only **c**. The rule says to do a recursion with **[c]** the first argument. But to solve that the second rule will do, so **M** is bound to **[c]**. So the answer is assembled from **X**, the word "censored", and that **M**, and the result for **Z** is **[a,censored,c]**.

Now consider an example list with four items:

```
?- mystery([a,b,c,d],L).
```

In the third rule, **X** is matched to **a**, **Y** is matched to **b**, and **[c,d]** is matched to **L**. The recursion applies mystery to the two-item list **[c,d]**, which by analogy to the preceding two-item example gives **[c,censored]**. So we assembled the answer from **X**, the word "censored", and the list **[c,censored]**, and the result for **Z** is **[a,censored,c,censored]**. So it looks like the program changes every alternate word in a list to the word "censored".

(b) One for even-length lists, one for odd-length. The rule lops off two items from the list on each recursion, so you can get into two different final situations.

5-19. (a) |N + 1| times, once for every item in the list plus once for the empty list; all these invocations will fail.

(b) |(N - 1) / 2|. There are 0 calls if the item is first in the list, one call if the item is second in the list, two calls if the item is third in the list, and so on. These terms form an arithmetic series. The average value of an arithmetic series is the average of the first and last numbers, in this case 0 and N - 1.

5-22. Use a list to keep arguments previously used, and just check to make sure a new value found is not in the list. At every recursion, add the new argument to the list. So:

```
a3(X,Y) :- a2(X,Y,[]).


a2(X,Y,L) :- a(X,Y).
a2(X,Y,L) :- not(member(X,L)), a(X,Z), a2(Z,Y,[X|L]).


member(X,[X|L]).
member(X,[Y|L]) :- not(X=Y), member(X,L).
```

Then you query **a3** to compute relationships by transitivity. For instance, suppose you have the database:

```
a(r,s).
a(s,t).
a(t,r).
```

(For instance, predicate **a** might mean "equals" or "is near".) Then the query

```
?- a(r,t).
```

will succeed as it should, as will

```
?- a(r,r).
```

But the query

```
?- a(r,u).
```

will fail as it should.

6-1.(a) Forward chaining means reasoning from facts to goals. The first fact is **c**, and it is mentioned in rules R5 and R8, so expressions in those rules are matched in that order. The right side of R8 is eliminated, so **b(X)** is made a new fact. (X isn't bound, but that's OK.) The "b(X)" goes at the front of the facts, so we now can match expressions in rules R1 and R3, but neither of those rules is eliminated yet.

So we pick the next fact **l**. This matches the right side in R7, and **g(X)** is made a new fact at the front of the fact list. This in turn matches an expression in R2. We next pick fact **e(a)** and this matches an expression in R5 only. So since **c** was already matched, R5 succeeds, and the fact **d** is asserted. But **d** is a goal, so we stop.

(b) Now we reason from goals to facts. The **f(X)** is the first goal (hypothesis), invoking R2, and we must prove **a(X)**. This in turn invokes R3, which invokes R8. R8 succeeds because **c** is a fact. But **i** is not a fact (nor are there any rules for it), so R3 fails. R3 is the only rule for proving **a(X)**, so R2 fails too.

So we try the next goal, **d**. R4 is the first applicable rule, and as before, **i** is not a fact nor provable, so R4 fails. We thus invoke R5 to prove **d**. The fact **e(a)** matches **e(X)**, and **c** is a fact, so **d** is proved.

(c) Yes, because if **c** and **j(b)** come before **e(a)** in the initial facts, the alternative goal **k(b)** will be proved first instead.

(d) No, fact order doesn't affect backward chaining unless there are facts with the same predicate name, which isn't true here. (If some of the facts had the same predicate name, different variables might be bound in the final goal, a problem that doesn't arise unless goals have variables.)

6-5. (a) **u**, **b**, **m(12)**, **a**. Steps in order:

1. Fact **r** creates new rules **t :- s.** and **u :- v.**

2. Fact **v** creates the new rule **a :- t**., and proves the fact **u** .

3. Fact **u** creates the new rule **a :- b, not(t).**

4. Fact **c** proves the fact **b**.

5. Fact **b** creates the new rules **a :- not(t).** and **m(X) :- n(X).**

6. Fact **n(12)** proves **m(12)**.

7. Fact **m(12)** proves nothing.

8. No facts remain unexplored, so we examine the rule with the **not**. Since **t** is not a fact, the **not** succeeds, and this proves **a**.

9. Fact **a** proves nothing.

(b) **b**, **u**, **m(12)**, **a**. Fact order doesn't matter here since all facts have different predicate names. Steps in order:

1. The first three rules fail because all mention at least one thing that doesn't match a fact.

2. The fourth rule succeeds since **c** is a fact, so the new fact **b** is asserted. The rule is deleted.

3. The fifth rule fails.

4. The sixth and last rule succeeds because **v** and **r** are both facts. So new fact **u** is asserted. The rule is deleted.

5. The first two rules again fail.

6. The third rule succeeds because **n(12)** and **b** are now facts. New fact **m(12)** is asserted. This rule is not deleted because it contains a variable on its left side.

7. No further rules succeed.

8. Now the **not**s are considered. In the second rule, **t** is not a fact, so **not(t)** succeeds, as well as **b** and **a**. So fact **a** is asserted.

9. No further rules succeed.

(c) Index the predicate names on right sides of rules, and index **not**'s. Then you need only check rules that contain predicate names of facts proved on the last cycle, and you can find **not**s fast. Also you can also delete rules that succeed whose left sides do not contain variables.

6-7. The minimum of L and S. Each cycle must prove something to keep things going. There are only L different things provable, so L is an upper bound. But each cycle must match also at least one new predicate name on a rule right side so that a new rule can succeed. So if there are S right-side names, S is an upper bound too.

6-8.(a) Backward chaining means reasoning from hypotheses to facts. Here the hypotheses are actions the robot may take, which we must consider in the order given. Hence the first hypothesis we examine is "turn around", and the rule order is:

> R1 fails, turn-around hypothesis fails, try stop-and-wait hypothesis, R3, R4, R16, R8 fails (no moving branches visible), R9 succeeds (so second object is an animal), R16 fails (four branches not present), R17 succeeds (so second object is a person), R4 succeeds, R3 fails (because robot isn't "beneath" anything), R5 succeeds (the only other way to achieve the stop-and-wait hypothesis, and it succeeds because we've already proved the second object is an animal).

You could also put R10 and R11 before first invocation of R16, if you interpreted "person or vehicle" in rule R4 as a single concept.

(b) Now we take the facts, in order, and reason about what other facts they imply:

> --Fact F1 matches expressions in rules R13, R14, and R16.
>
> --Fact F2 matches expressions in rule R13.
>
> --Fact F3 matches expressions in rule R12.
>
> --Fact F4 matches expressions in rule R12, and that rule is completely satisfied. Hence the first object is an obstacle, and call that fact F4.1.
>
> --Fact F4.1 matches expressions in rules R6, R13, R14, and R15. Now R13 is completely satisfied. Hence the object to the right is a bush, and call this fact F4.2.
>
> --Fact F4.2 matches expressions in rules R2 and R3.
>
> --Fact F5 matches expressions in rules R4 and R5.
>
> --Fact F6 matches expressions in rules R9 and R17, and R9 is satisfied. Hence the second object is an animal, and call that fact F6.1.
>
> --Fact F6.1 matches expressions in R16 and R17, and R17 is completely satisfied. Hence the

second object is a person, and call that fact F6.2.

--Fact F6.2 matches expressions in rule R4, and that rule is completely satisfied. Hence we must hide, and call that fact F6.3.

--Fact F6.3 matches expressions in rule R2, and that rule is completely satisfied. Hence we turn towards the bush and move a short distance.

(c) The backwards chaining used a fixed-rule-priority conflict resolution, and forward chaining used a fixed-fact-priority (with focus-of-attention) in addition to fixed-rule-priority. An alternative for forward chaining would be to add new facts at the end of the facts, while we still fetch from the front as with a queue. This would be good when we want to infer everything possible from a set of facts. An alternative for backward chaining would be a "biggest-rule-first" strategy in which the rule with the most conditions in its "if" part is tried first. Concurrency and partitioning ideas could work for both forward and backward control structures. We could also use a "most-related-rule" criterion, if we define "relatedness" for any rule pair.

6-9. (a) No, since a conclusion may depend on the absence of some fact. That fact may no longer be absent when a file is loaded in later.

(b) Yes, because caching just makes things already found easier to verify. Provability of conclusions is not affected, just efficiency.

(c) No, since new facts could be asserted that are unreachable by backward chaining. So a conclusion that depends on the absence of something might suddenly go from being true to being false.

6-12. (a) Forward chaining sounds better, since the system only need reason about sensor readings that changed from its last analysis (assuming we cache the conclusions from the last analysis), and changes to sensor readings will be relatively rare. And probably a lot of different goals (conclusions) can be reached.

(b) It's true that forward chaining caches automatically, so extra caching would seem unnecessary. But the best way to use this rule-based system is to run it repeatedly every second or so. Then conclusions learned during the last second could be useful in reasoning during this second--especially conclusions based on things that don't change often--so caching the previous conclusions could be useful. Also, caching could mean storing (in the rule-based system database) facts about the condition of sensors instead of directing the computer to look at its input/output ports to find those values. This also will be good because many sensors will infrequently change with time.

(c) Virtual facts don't make sense with forward chaining. If you suggested backward chaining in part (a), virtual facts might make sense if you interpret them as "queries" to sensor input/output ports. If you interpret them as queries to the user, they don't make sense because that would be an big imposition on the user.

(d) Yes, since there are clear categories of rules that don't interact much. For instance, the rules for handling burglars might not be in the database all the time, but only loaded when there is good evidence that a burglar is present. Rules for fires and other emergencies are similar. However, the rules for *recognizing* potential burglar and fire situations, that decide to load the detail-analyzing code, must be kept in main memory at all times.

(e) This is a closer judgment than the other parts of this question, but decision lattices are probably not a good idea. In the first place, you don't want to use them in a way so that you actually ask a human questions--that would be an imposition. But the decision lattice could "ask" questions of the sensors to get their readings. The main difficulty is in building the decision lattice in the first place. There will be many different situations to handle, and it's hard to figure out what questions distinguish situations well, especially when houses are all different and you don't know what situations you'll get into; and it's the very rare emergency situations that are most critical to program properly. Furthermore, decision lattices are hard to modify, and hard to partition into modules. So for the advantages of a decision lattice, it would seem better to go with an and-or-not lattice, for which the translation from rules to lattice is much easier.

(f) Yes. Improved speed in inferencing is an obvious advantage over noncompiled forms. And-or-not lattices have the advantage of being easy to partition if the original rule-based system was easy to partition, like this one; partitioning a decision lattice is much harder, and since this application can easily be complex, inability to partition into modules is a serious disadvantage. And-or-not lattices are easier to modify than decision lattices, since they correspond closely to rules. And-or-not lattices could also standardize certain commonly-made inferences, so these could be mass-produced and a smaller, more house-specific computer could handle the rest of the reasoning. For instance, the conditions for suspecting a burglar and loading burglar-analysis rules are pretty much standard for all houses and with all burglars: check for unusual noises and for windows and doors being forced open, especially at night and when the house is not occupied. As another example, equipment requiring complicated continuous monitoring like an aquarium could be monitored according to a special and-or-not lattice supplied by the aquarium manufacturer, and only reports of major problems passed on to the central processor. It's true that and-or-not lattices have trouble handling variables, but this rule-based system doesn't much need them: use no-argument predicates to represent sensor values and value ranges.

7-1. Here's one way:

```
diagnosis('Fuse blown') :- power_problem, askif(lights_out).
diagnosis('Fuse blown') :- power_problem, askif(hear(pop)).
diagnosis('Break in cord') :- power_problem, askif(cord_frayed).
diagnosis('Short in cord') :- diagnosis('fuse blown'),
  askif(cord_frayed).
diagnosis('Device not turned on') :- power_problem, klutz_user,
  askif(has('an on-off switch or control')), askifnot(device_on).
diagnosis('Cord not in socket properly') :- power_problem,
  klutz_user, askif(just_plugged), askifnot(in_socket).
diagnosis('Internal break in the wiring') :-
  power_problem, jarring.
diagnosis('Foreign matter caught on heating element') :-
  heating_element, not(power_problem), askif(smell_smoke).
diagnosis('Appliance wet--dry it out and try again') :-
  power_problem, klutz_user, askif(liquids).
diagnosis('Controls adjusted improperly') :- klutz_user,
  askif(has('knobs or switches')).
```

```
diagnosis('Motor burned out') :- askif(smell_smoke),
  mechanical_problem.
diagnosis('Something blocking the mechanical operation') :-
  mechanical_problem.
diagnosis('Kick it, then try it again') :- mechanical_problem.
diagnosis('Throw it out and get a new one').


power_problem :- askif(device_dead).
power_problem :- askif(has('knobs or switches')),
  askifnot(knobs_do_something).
power_problem :- askif(smell_smoke), not(heating_element).


klutz_user :- askifnot(handyperson).
klutz_user :- askifnot(familiar_appliance).


mechanical_problem :- askif(hear('weird noise')),
  askif(has('moving parts')).


heating_element :- askif(heats).
heating_element :- askif(powerful).


jarring :- askif(dropped).
jarring :- askif(jarred).

questioncode(device_dead,'Does the device refuse to do anything').
questioncode(knobs_do_something,'Does changing the switch
  positions or turning the knobs change anything').
questioncode(lights_out,
  'Do all the lights in the house seem to be off').
questioncode(code_frayed,
  'Does the outer covering of the cord appear to be coming apart').
questioncode(handyperson,'Are you good at fixing things').
questioncode(familiar_appliance,
  'Are you familiar with how this appliance works').
questioncode(device_on,'Is the ON/OFF switch set to ON').
questioncode(just_plugged,'Did you just plug the appliance in').
questioncode(in_socket,'Is the cord firmly plugged into the socket').
questioncode(smell_smoke,'Do you smell smoke').
questioncode(liquids,
  'Have any liquids spilled on the appliance just now').
```

```
questioncode(heats,'Does the appliance heat things').
questioncode(powerful,'Does the appliance require a lot of power').
questioncode(has(X),X) :- write('Does the appliance have ').
questioncode(hear(X),X) :- write('Did you hear a ').
questioncode(dropped,'Has the appliance been dropped recently').
questioncode(jarred,'Has the appliance been violently jarred recently').
```

7-2. Use the same idea of **coded_diagnosis** at the end of Section 7.4. Query instead of **diagnosis**:

```
better_diagnosis(D) :- diagnosis(D), not(found(D)), asserta(found(D)).
```

Now if the user types a semicolon after a diagnosis, the program will backtrack through the **asserta** and **found** to try to find another, but if D gets rebound to any previous value, the **not** will fail and the interpreter returns to "diagnosis". This also works for any order of diagnosis rules; rules for the same diagnosis need not be together.

7-5. (a) One unit of time is needed if the first cached item matches the query, two units of time if the second matches the query, three units of time if the third, and so on. Since we were told the probabilities are mutually exclusive, the probability that no item in the cache matches the query is |1 - KP|. Then the condition for the problem is

```
P + 2P + 3P + ... + KP + ( 1 - KP ) ( R + K )    <   R
```

The left side of the inequality contains an arithmetic series, so we can substitute its formula and get

```
P K ( K + 1 ) / 2   +   ( 1 - KP ) K   <    KPR
```

or

```
( K + 1 ) / 2   +   ( 1 / P )   -   K   < R
```

```
( 2 + P - PK ) / 2P   <   R
```

Since |KP < .1|, it's also true that |P < .1|, so we can approximate the inequality by just

```
PR > 1
```

(b) Analysis is similar, but the sum over the cache items has a different formula:

```
P + ( 2 * P / 2 ) + ( 3 * P / 3 ) + ... ( K * P / K )   =   KP
```

so the criterion becomes

```
KP + ( R + K ) ( 1 - P log sub 2 ( K + 1 ) )   <   R
```

or

```
K ( P + 1 )   <   P ( R + K ) log sub 2 ( K + 1 )
```

If |PK < .1| then because |P < PK| we know P is negligible compared to 1, so we can approximate this by

```
1    <   P ( ( R / K ) + 1 ) log sub 2 ( K + 1 )
```

7-11. (a)

```
or(P,Q) :- call(P).
or(P,Q) :- call(Q).
```

(b)

```
if(P,Q,R) :- call(P), call(Q).
if(P,Q,R) :- not(call(P)), call(R).
```

(c)

```
case(P,N,L) :- call(P), item(N,L,I), call(I).


item(1,[X|L],X).
item(N,[X|L],I) :- N>1, N2 is N-1, item(N2,L,I).
```

8-3.(a) So **<prob1>** is |200 / 500 = 0.4|, and **<prob2>** is |800 / 1600 = 0.5|. The 1200 flat tires are irrelevant.

(b) The ratio 70/101 is approximately 0.69, and this is closest to 0.7, the independence-assumption combination (obtained from |1 - (1 - 0.4) (1 - 0.5)|). Conservative assumption gives |"maxfunction" (0.4 , 0.5) = 0.5|, and liberal gives |0.4 + 0.5 = 0.9|.

8-4.(a) conservative, since the function is the maximum of the two arguments.

(b) liberal. It can't be conservative since the values in the table are better than the previous table. It can't be independence-assumption because the combination of two "probably"s is a "definitely", something impossible with the independence formula (that is, the independence formula can't give a 1.0 unless one of its inputs is 1.0). The arithmetic (see part (c)) is consistent with liberal combination too.

(c) Let |x| be the probability of "probably not", |y| the probability of "probably". Then the table and the liberal-combination formula say that |x + x = y| and |minfunction (x + y , 1.0) = 1.0 |. So |2x = y| and |minfunction (

3x , 1.0) = 1.0 |. Hence |x > 0.3333| from the last equation. But |y| must be less than 1.0 ("probably" should be less certain than "definitely") so |2x < 1.0| and |x < 0.5|. So |x| can be anything between 0.3333 and 0.5, and |y| must be correspondingly between 0.6667 and 1.0. (Notice that an infinity of possible answers doesn't mean *any* answer is right.)

8-5. No. There is no problem with either backward chaining, forward chaining, or any hybrid. Facts must always have probability values filled in (or else be certain, implying a probability of 1.0), so any rules combining those probabilities will work no matter how they're used.

8-6.(a) You must be careful to count separately the length of the list and the number of probabilities greater than 0.5. One way to do it:

```
orcombine(PL,P) :- numbigprobs(PL,Nbig), length(PL,N), P is Nbig / N.


numbigprobs([],0).
numbigprobs([P|PL],N) :- numbigprobs(PL,N), P < 0.5.
numbigprobs([P|PL],N) :- numbigprobs(PL,N2), P >= 0.5, N is N2 + 1.


length([],0).
length([X|L],N) :- length(L,N2), N is N2 + 1.
```

Another way is to define a **censor** predicate that removes probabilities from a list that are less than 0.5:

```
orcombine(PL,P) :- length(PL,N), censor(PL,CPL), length(CPL,Nbig),
  P is Nbig / N.


length([],0).
length([X|L],N) :- length(L,N2), N is N2 + 1.


censor([],[]).
censor([P|PL],[P|PL2]) :- censor(PL,PL2), P >= 0.5.
censor([P|PL],PL2) :- censor(PL,PL2), P < 0.5.
```

(b) Some answers:

> --There are abrupt changes in the total probability when probabilities vary slightly from 0.5. It doesn't seem reasonable that any evidence combination method have such abrupt "jumps" as probabilities vary slightly.
>
> --The number 0.5 is arbitrary--there's no good reason for it.

> --The total probability with this method can be less than the "conservative" combination value; for example, **[0.4,0.4]** has 0 total with the method, 0.4 with the "conservative" method. But the conservative method gives the smallest logically reasonable value, so this method isn't always reasonable.
>
> --This method is inaccurate when few probabilities are combined. If there's only one probability in the list presented for combination, then the cumulative probability is either 0 or 1, totally ignoring the probability of that one item.
>
> --This method can't handle "cascades" very easily. That is, if the result of this method is an input to a combination in another rule, we've lost the important information about the number of probabilities contributing to the first result, which should greatly affect how much it should be counted for in the second combination.

8-9. (a) Newspapers and television report unusual and striking events, so you don't hear about the many more people that spent their money on the lottery and didn't win anything.

(b) In this example, successes in the lottery are broadcast much more strongly than failures, causing overestimation of the probability of success. This "biased sampling" problem is a chief difficulty in building rule-based systems from data: you must be careful there aren't reasons why certain data is overreported or underreported. For instance, embarassment might make people underestimate the frequency with which they forget to plug appliances in.

9-2. For inheritance, the goal condition is to find some other target object that has the property value you want, and has an inheriting link, like **a_kind_of**, to a given object. Looking for such a target object is a search starting from the given object. There can be several inheriting links for an object, each representing an alternative direction to explore, and these represent branches. And one choice affects later choices.

9-10. Just subtract K from the values of the old evaluation function to get a new evaluation function which is guaranteed to be a lower bound. Then you can use A* search with the optimality guarantee.

9-11.(a) See Figure G-1.

(b) Choose the state (among those so far discovered at some point in the problem) that has lowest evaluation function value. State sequence: a, d, c, b, f, e, g, h.

(c) Add the cost to reach a state to the evaluation function. Costs to states: a: 0, b: 2, c: 5, d: 9, e: 7, f: 11, g: 7, h: 11. Sum of cost and evaluation function: a: 10, b: 10, c: 12, d: 14, e: 13, f: 15, g: 16, h: 25. State sequence: a, b, c, e, d, f, g, h.

9-13. (a) All **[<left-bank>,<right-bank>]**, where **<left-bank>** and **<right-bank>** are lists together containing the symbols man, lion, fox, goose, corn, and boat. And **<left-bank>** is things on the left side of the river, **<right-bank>** things on the right.

(b) The start is **[[man,lion,fox,goose,corn,boat],[]]** and goal is **[[],[man,lion,fox,goose,corn,boat]]**.

(c) Man and boat; man, lion, and boat; man, fox, and boat; man, goose, and boat; man, corn, and boat; man, fox, corn, and boat; or man, goose, corn, and boat move from one side of the river to the other.

(d) The starting state has only one successor, **[[lion,goose],[man,fox,corn,boat]]**, which has three successors of its own (besides the starting state):

```
[[man,lion,goose,boat],[fox,corn]].
[[man,lion,fox,goose,boat],[corn]].
[[man,lion,goose,corn,boat],[fox]].
```

(e) |(1 + 3) / 2 = 2|

(f) Each thing can be in one of two places, and there are five things (the man must always be with the boat), hence |2 sup 5 = 32|.

(g) No, subparts interact.

9-15.(a) The set of all possible coating patterns.

(b) No, there are many different states (coating patterns) that satisfy the goal conditions, and no easy way to find them. (If you knew any of them, you wouldn't need to search.) So backward search isn't possible, and hence bidirectional search isn't either. (Notice that despite similarities, this problem is quite different from city-route planning: the goal is a configuration of coatings, not a location.)

(c) It always decreases. Every branch involves coating a grid cell, thus removing it from the pool of coatable cells. As you proceed, previously-coatable cells may become uncoatable because they would connect coated regions that shouldn't be connected, so the branching factor may decrease by more than one per level, but it will always decrease.

(d) Many heuristics are possible, including:

> --Prefer to coat cells adjacent to the last cell you coated (a "focus of attention" heuristic).
> --Prefer to coat cells that lie on a straight line between points you want to connect.
> --As soon as a desired connection is made between two points, prefer not to coat any cells within five cells of either of those points.
> --Prefer to coat cells that continue a straight line (e.g., coat [X,Y] if [X-1,Y] and [X-2,Y] are both coated).

10-1.(a) Change the definition of **depthsearch** to read

```
depthsearch(Start,Ans) :- depthsearch2(Start,[Start],Ans),
  print_reverse(Ans).
```

and include the definition

```
print_reverse([]) :- !.
print_reverse([X|L]) :- print_reverse(L), write(X), nl.
```

(b) Add to the front of all state lists the name of the last operator used to reach that state (or "none" for the starting state). (This assumes that states are represented by lists, as in most applications.) Then modify all **successor**, **goalreached**, **eval**, and **cost** predicate definitions to ignore the first term of the input state description, and modify all **successor** predicate definitions to set the first item of the output successor state to an operator name appropriate to the purpose of each such rule or fact.

(c) Replace the **member** in **not(member(Newstate,Statelist))** by **permutedmember**, defined as:

```
permutedmember(L1,L2) :- subset(L1,L2), subset(L2,L1), !.
subset([],[]) :- !.
subset([X|L1],L2) :- member(X,L2), !, subset(L1,L2).
```

where **member** is defined as before.

10-5. Refer to states as a pair of numbers representing fluid quantities in the size-5 and size-7 glasses respectively. So [3,4] means 3 units in the size-5 glass and 4 units in the size-7 glass.

(a) The states form a circle (excluding [5,7] which is only reachable when the goal amount is 5 or 7). Two solutions are always found for goal amounts 1, 2, 3, 4, and 6, one from going around the circle clockwise and one counterclockwise, both stopping at the first state satisfying the goal condition. (To see this behavior, it's important to avoid duplicating the same state at different places in the search graph.)

(b) The breadth-first state graph is augmented by extra links to the states [5,0], [0,7], and [5,7]. Links to the first occur from states [5,X] and [X,0] for certain X, to the second from [X,7] and [0,X] for certain other X, and to the third from [X,7] and [5,X] for certain other X. On such transitions, the search "starts over" in the opposite direction around the circle (so if movement was clockwise before the transition, afterwards it is counterclockwise). Only one such transition can occur in a solution path, except when the goal amount is 5 or 7.

(c) The eight rules make four pairs, for which the first rule of each pair refers principally to the first glass, the second rule to the second glass. The first pair empties one glass into another, the second pair fills one glass from another, the third pair fills a glass from the faucet, and the fourth pair empties a glass into the drain.

(d) Delete the second **goal_reached** fact.

10-7. Branch-and-bound search is just an A* search in which the evaluation function is always zero. So just to use the definition

```
eval(S,0).
```

with the A* program. That will work fine if the problem-independent file is always loaded before the problem-dependent file. But if you can't be sure, you can remove the **eval** predicate from the **add_state** predicate definition:

```
add_state(Newstate,Pathlist) :- not(agenda(Newstate,P,C)),
  not(oldagenda(Newstate,P2,C2)), cost([Newstate|Pathlist],Cnew),
  asserta(agenda(Newstate,[Newstate|Pathlist],Cnew)).
```

10-10. (a) Represent states as a list of pairs, where the first element of each pair is the name of a chemical and the second element is its amount in moles. Define **increase** and **remove** predicates that change the amount of a chemical in a state list. (Write "remove" to fail when it can't find the specified chemical in the state, so you don't need a **member** predicate.)

```
go(Ans) :- search([[cl2,2],[mno2,1],[caco3,1],[h2o2,2]],Ans).


goalreached(S) :- member([cacl2,N],S), N >= 1.


successor(S,[[C,Nsum]|S3]) :- remove(C,N1,S,S2),
  remove(C,N2,S2,S3), Nsum is N1 + N2.
successor(S,S6) :- remove(cl2,Ncl2,S,S2), remove(h2o,Nh2o,S2,S3),
  N is Ncl2 - Nh2o, N >= 0, increase(cl2,N,S3,S4),
  increase(hclo,Nh2o,S4,S5), increase(hcl,Nh2o,S5,S6).
successor(S,S6) :- remove(cl2,Ncl2,S,S2),
  remove(h2o,Nh2o,S2,S3), N is Nh2o - Ncl2, N >= 0,
  increase(h2o,N,S3,S4), increase(hclo,Ncl2,S4,S5),
  increase(hcl,Ncl2,S5,S6).
successor(S,S4) :- remove(h2o2,N,S,S2), member([mno2,X],S2),
  HalfN is N / 2, increase(h2o,N,S2,S3),
  increase(o2,HalfN,S3,S4).
successor(S,S6) :- remove(caco3,Ncaco3,S,S2),
  remove(hcl,Nhcl,S2,S3), HalfNhcl is Nhcl/2, Ncaco3 >= HalfNhcl,
  increase(cacl2,HalfNhcl,S3,S4), increase(co2,HalfNhcl,S4,S5),
  increase(h2o,HalfNhcl,S5,S6).
successor(S,S6) :- remove(caco3,Ncaco3,S,S2),
  remove(hcl,Nhcl,S2,S3), HalfNhcl is Nhcl/2, Ncaco3 < HalfNhcl,
  increase(cacl2,Ncaco3,S3,S4), increase(co2,Ncaco3,S4,S5),
  increase(h2o,Ncaco3,S5,S6).
successor(S,S5) :- remove(h2,Nh2,S,S2), remove(cl2,Ncl2,S2,S3),
  Nh2 >= Ncl2, N is Nh2-Ncl2, increase(h2,N,S3,S4),
  TwiceNcl2 is 2*Ncl2, increase(hcl,TwiceNcl2,S4,S5).
```

```
successor(S,S5) :- remove(h2,Nh2,S,S2), remove(cl2,Ncl2,S2,S3),
  Nh2 < Ncl2, N is Ncl2-Nh2, increase(cl2,N,S3,S4),
  TwiceNh2 is 2*Nh2, increase(hcl,TwiceNh2,S4,S5).
successor(S,S7) :- remove(hcl,Nhcl,S,S2),
  remove(mno2,Nmno2,S2,S3), QuarterNhcl is Nhcl/4,
  QuarterNhcl >= Nmno2, N is QuarterNhcl - Nmno2,
  increase(hcl,N,S3,S4), TwiceNmno2 is 2*Nmno2,
  increase(mncl2,Nmno2,S4,S5), increase(h2o,TwiceNmno2,S5,S6),
  increase(cl2,Nmno2,S6,S7).
successor(S,S7) :- remove(hcl,Nhcl,S,S2),
  remove(mno2,Nmno2,S2,S3), QuarterNhcl is Nhcl/4,
  QuarterNhcl < Nmno2, N is Nmno2 - QuarterNhcl,
  increase(mno2,N,S3,S4), HalfNhcl is Nhcl/2,
  increase(mncl2,QuarterNhcl,S4,S5),
  increase(h2o,HalfNhcl,S5,S6), increase(cl2,Nmno2,S6,S7).

increase(Chemical,Amount,S,[[Chemical,Newamount]|NewS]) :-
  failing_delete([Chemical,Oldamount],S,NewS),
  Newamount is Oldamount + Amount, !.
increase(Chemical,Amount,S,[[Chemical,Amount]|S]).



remove(Chemical,Amount,S,NewS) :-
  failing_delete([Chemical,Amount],S,NewS).



failing_delete(X,[X|L],L) :- !.
failing_delete(X,[Y|L],[Y|L2]) :- failing_delete(X,L,L2).
```

(b)

```
?- go(Answer).
(11) 11 Call: successor([[cl2,2],[mno2,1],[caco3,1],[h2o2,2]],_8)
(11) 11 Back to: successor([[cl2,2],[mno2,1],[caco3,1],[h2o2,2]],_8)
(11) 11 Back to: successor([[cl2,2],[mno2,1],[caco3,1],[h2o2,2]],_8)
(11) 11 Back to: successor([[cl2,2],[mno2,1],[caco3,1],[h2o2,2]],_8)
(11) 11 Exit: successor([[cl2,2],[mno2,1],[caco3,1],[h2o2,2]],
  [[o2,1],[h2o,2],[cl2,2],[mno2,1],[caco3,1]])
(63) 19 Call:
  successor([[o2,1],[h2o,2],[cl2,2],[mno2,1],[caco3,1]],_67)
(63) 19 Back to:
  successor([[o2,1],[h2o,2],[cl2,2],[mno2,1],[caco3,1]],_67)
(63) 19 Exit:
  successor([[o2,1],[h2o,2],[cl2,2],[mno2,1],[caco3,1]],
  [[hcl,2],[hclo,2],[cl2,0],[o2,1],[mno2,1],[caco3,1]])
```

```
(112) 28 Call:
  successor([[hcl,2],[hclo,2],[cl2,0],[o2,1],[mno2,1],[caco3,1]],_123)
(112) 28 Back to:
  successor([[hcl,2],[hclo,2],[cl2,0],[o2,1],[mno2,1],[caco3,1]],_123)
(112) 28 Back to:
  successor([[hcl,2],[hclo,2],[cl2,0],[o2,1],[mno2,1],[caco3,1]],_123)
(112) 28 Back to:
  successor([[hcl,2],[hclo,2],[cl2,0],[o2,1],[mno2,1],[caco3,1]],_123)
(112) 28 Back to:
  successor([[hcl,2],[hclo,2],[cl2,0],[o2,1],[mno2,1],[caco3,1]],_123)
(112) 28 Exit:
  successor([[hcl,2],[hclo,2],[cl2,0],[o2,1],[mno2,1],
  [caco3,1]],[[h2o,1],[co2,1],[cacl2,1],[hclo,2],[cl2,0],[o2,1],[mno2,1]])

Answer=[[[h2o,1],[co2,1],[cacl2,1],[hclo,2],[cl2,0],[o2,1],[mno2,1]],
       [[hcl,2],[hclo,2],[cl2,0],[o2,1],[mno2,1],[caco3,1]],
       [[o2,1],[h2o,2],[cl2,2],[mno2,1],[caco3,1]],
       [[cl2,2],[mno2,1],[caco3,1],[h2o2,2]]]

yes
```

(c) Cost could be the monetary cost of the chemicals needed for all the reactions so far, and the evaluation function could be the average cost of a chemical times the number of chemicals not yet in the reaction vessel but desired in the goal state. Or cost could be the danger of doing a sequence of reactions, and the evaluation function an estimate of the remaining danger based on the goal state.

(d) In the real world, all possible chemical reactions occur simultaneously. To model this for our rules, all that apply should work in parallel somehow. A little of each reaction could be done, time-sharing between reactions cyclically. (To be more precise, reactions should proceed at different speeds determined by *equilibrium constants* for each reaction, which can be modeled by "biased" time-sharing.).

10-12. (a) 16 states, if the goal state is on the final (leaf) level.

(b) 16 as well. With best-first search, the agenda can contain states at many levels. But if it does contain states at levels other than level 4 (the leaf level) then we could substitute in the successors of those states to get a bigger possible agenda. Therefore the largest agenda must contain only level 4 (leaf states), and there are 16 of those.

10-15. This is what the **foriterate** predicate definition does, so just query:

```
call(P,K) :- foriterate(call(P),K).
```

So just use the two-argument version of **call** instead of the usual one-argument form. If for instance you wanted to print out the third value of **X** to which the expression **a(X)** can be matched, you would query:

```
?- call(a(X),3), write(X).
```

11-1. These correspond to the **recommended** definitions. And rules for such household hints should go in front of default **recommended** rules for the same situations.

11-4. You need new predicates to describe the condition of the plate--let's say **on** and **off** with argument "plate". A starting state for the testing can be

```
[closed(case),closed(top),inside(batteries),defective(batteries),
  defective(light),unbroken(case),on(plate)]
```

with the goal

```
[ok(batteries),ok(light),closed(case),closed(top)]
```

Define new operators **remove_plate** and **replace_plate** this way:

```
recommended([off(plate)],remove_plate).
recommended([on(plate)],replace_plate).


precondition(remove_plate,[open(top),on(plate)]).
precondition(replace_plate,[open(top),off(plate)]).


deletepostcondition(remove_plate,[on(plate)]).
deletepostcondition(replace_plate,[off(plate)]).


addpostcondition(remove_plate,[off(plate)]).
addpostcondition(replace_plate,[on(plate)]).
```

To relate the new predicates and operators to the code written before, modify the preconditions of **replace_light** and **assemble_top**:

```
precondition(replace_light,[off(plate),open(top)]).
precondition(assemble_top,[on(plate),open(top)]).
```

11-6. Means-ends analysis is recursive, and recursion requires a stack when implemented in a computer. Laying out the parts in the order you remove them from the car is like keeping a stack, so you can "pop" them--that is, put them back into the car--in the reverse of the order they came out. It's not fair to say the parts are preconditions to operators, however; they're just props used in achieving preconditions, and the real preconditions are abstract notions.

11-7. Just write two different **precondition** facts, one for each set of preconditions. Put the set you think more likely to succeed first. If it can't be satisfied, backtracking will take the second set of preconditions. Nothing we've said requires a unique **precondition** rule for each operator.

12-3.(a) A value inherited must be for a slot that's filled in for the "purchase order" frame. Unfortunately, the frame is quite abstract, so most of the slots that you could imagine for it wouldn't have values (though those slots themselves will inherit downward via slot inheritance). One answer would be a **purpose** slot in a purchase order, always filled in with the value "purchasing_something". Another answer would be a **where_obtained** slot always filled with value "stockroom", meaning you can get blank purchase orders from the stockroom.

(b) It inherits from P2 too. (That doesn't necessarily mean that P2 has its units slot filled in--P2 could be inheriting that value itself. But that's still "inheritance from P2".)

12-4. Use frames to represent slots themselves. Have the name of each possible slot be the name of a slot-defining frame, in which we have a special slot named **value** to hold the value. Each slot-defining frame can have a slot for each qualifying slot of the original slot, with values filled in as appropriate. Every time you refer to the slot name, you can just inherit information from its slot-defining frame.

12-5. (a) The value "Floodge Manufacturing Inc." in the manufacturer's name slot, since it must be filled in similarly in any subtype of computer made by Floodge.

(b) The slot **ID-number**, since every computer must have an ID number, but each computer has a different value for it.

(c) Every Floodge computer has a CPU. So every personal computer made by Floodge also has a CPU. The CPU of a personal computer made by Floodge is an example of a part-kind inheritance, inference of a new frame. This new frame is different from the CPU frame for all Floodge computers, because for instance personal computers have slower CPUs.

12-9. The relationship of the Navy to military-organizations is **a_kind_of**, whereas the relationship of NPS to the Navy is more like **part_of**. Or you could say the Navy and military organizations are intensions, while NPS is more like an extension.

12-13. The problem of multiple inheritance for qualifying slots should be simpler than the original multiple inheritance problem, since they usually have few different values. So you can just assign priorities of inheritance direction as suggested for the user models example. In other words, any "qualifying-qualifying" slot for inheritance method will be filled with the same value, this direction-priority method.

13-1. We have two orderings to deal with: the order of the three rules, and the order of the two terms in the first rule. The second order involves an "and" and these should be ordered with the thing most likely to fail first. So **c(X)** should come before **b(X)** since 0.7 is less than 0.8. Since we're told that all the probabilities are independent, the probability of that first rule succeeding is just the independence-assumption "andcombine" of 0.7 and 0.8, or 0.56. This 0.56 is greater than 0.1, but less than 0.6. So since we should put the rules with greatest probability of success first, we should put the **e** rule first, then the rule with **b** and **c**, and then the rule with **d**.

13-4.(a) There's a mistake or bug somewhere, because the existence of the picture implies there must be some possible interpretation of it. Maybe our constraints are unreasonable, or maybe the characteristics of the regions of the picture were described incorrectly.

(b) This suggests an optical illusion, a situation for which the human eye could see two possible interpretations. Optical illusions can arise at random, particularly with pictures of geometrical figures and pictures in which objects have lined up in unusual ways.

(c) This suggests not enough constraints or too-weak constraints to solve the problem. That's not necessarily the programmer's fault, many real-world pictures can give this behavior, like those with few sharp boundaries.

13-6. We must first create possibility lists, using the three meal types given and any single-variable constraints. The first constraint is the only single-variable one. So we have the following possibilities (using "m1" for meal 1, "B1" for breakfast on day one, "L2" for lunch on day two, etc.)

```
B1: [m1]
L1: [m1,m2,m3]
D1: [m1,m2,m3]
B2: [m1,m2,m3]
L2: [m1,m2,m3]
D2: [m1,m2,m3]
```

Now we do the multivariable part of relaxation starting with the B2 variable. (The instructions also suggest that we start with B2 in the first step, but that doesn't make the difference to the result that it does for this step.) We retrieve the possible values of a variable, and check which of them satisfy constraints C2, C3, and C4. Here is one way:

1. Pick variable B2. Possibility m1 is impossible by constraint 4.

2. Pick variable L2 by the focus-of-attention heuristic. Then m1 is impossible by constraint 4.

3. Pick L1 by the same heuristic. It cannot be m1 by constraint 2. And it can't be m3 be constraint 4. So it must be m2. Possibility lists are now:

```
B1: [m1]
L1: [m2]
D1: [m1,m2,m3]
B2: [m2,m3]
L2: [m2,m3]
D2: [m1,m2,m3]
```

4. Pick variable L2. Then m2 can be eliminated by constraint 4 so the only remaining possibility is m3.

5. Pick B2. It cannot be m3 by constraint 3 so it must be m2.

6. Pick D1. It cannot be m2 by constraint 2, and it cannot be m3 by constraint 3 so it must be m1.

7. Pick D2. It cannot be m3 by constraint 3, and it cannot be m1 be constraint 1,so it must be m2.

The final possibilities have been reduced to one for each variable, as follows: B1=m1, L1=m2, D1=m1, B2=m2, L2=m3, D2=m2.

13-13. (a) 162 times, to get the answer **X=4, Y=3, Z=5**:

1. 3 backtracks from **n(Y)** to **n(X)**, since four **X** values must be tried.

2. 17 backtracks from **n(Z)** to **n(Y)**, |(3 * 5) + 3 - 1|. (The five values for **Y** must be cycled through three times, then the third value succeeds.)

3. 89 backtracks from **X>Y** to **n(Z)**, |(17 * 5) + 5 -1|. (Similar reasoning to the preceding.)

4. 29 backtracks from the last predicate expression **g(X,Y,Z)** to **not(f(X,X))**, | (6 * 5) - 1 |. (Six (X,Y) pairs are tried: (2,1), (3,1), (3,2), (4,1), (4,2), and (4,3).)

5. 24 backtracks from **not(f(X,X))** to **X>Y**, |29 - 5|. (The same as the previous answer except for the **X=2** and **Y=1** case ruled out.)

(b) The **g** predicate is the hardest to satisfy, and the > comparison is next hardest, so write the query as

```
g(X,Y,Z), X>Y, not(f(X,X)), n(X), n(Y), n(Z).
```

This will backtrack a mere three times: all three from the **X>Y** to **g(X,Y,Z)**. The same answer is found.

(c) 39 times. First figure out where to backtrack to. For predicate expression P, backtrack to the last predicate expression previous to P that binds a variable in P, or the immediately previous expression if no expression binds a variable in P.

--n(Y) should backtrack to **n(X)**
--n(Z) should backtrack to **n(Y)**
--X>Y should backtrack to **n(Y)**
--not(f(X,X)) should backtrack to **n(X)**

--**g(X,Y,Z)** should backtrack to **n(Z)**

Let's now simulate dependency-based backtracking:

1. We match **X=1**, **Y=1**, and **Z=1**.

2. **X>Y** fails; we go back to **n(Y)**. (1 backtrack)

3. We pick **Y=2**, then return to **X>Y**, which fails again. Back to **n(Y)**. (1 backtrack)

4. We pick **Y=3**, then return to **X>Y**, which fails again. Back to **n(Y)**. (1 backtrack)

5. This repeats until **Y=5**, whereon we fail **n(Y)**. We return to **n(X)** and pick **X=2**. (3 backtracks)

6. We now pick **Y=1**, and **X>Y** succeeds.

7. But **f(X,X)** succeeds, so **not(f(X,X))** fails, so we backtrack to **n(X)**. (1 backtrack)

8. We now take **X=3**. We skip **n(Y)** and **n(Z)** since they could not have caused the last failure.

9. **X>Y** holds, as does **not(f(X,X))**. But **g(X,Y,Z)** fails, so we backtrack to **n(Z)**. (1 backtrack)

10. **Z=2** is chosen, we return to **g(X,Y,Z)**, but that doesn't work either. We return to **n(Z)**. (1 backtrack)

11. We alternate repeatedly between **n(Z)** and **g(X,Y,Z)** until **n(Z)** fails, whereupon we return to **n(Y)**. (4 backtracks)

12. We pick **Y=2**, **Z=1**. **X>Y** succeeds, but **g(X,Y,Z)** fails. Return to **n(Z)**. (1 backtrack)

13. None of the other values for **Z** will make **g(X,Y,Z)** succeed. Fail **n(Z)**. (5 backtracks)

14. We pick **Y=3**. **X>Y** fails. Similarly for **Y=4** and **Y=5**. So **n(Y)** fails. (4 backtracks)

15. We pick **X=4**, **Y=1**, **Z=1**. **X>Y** and **not(f(X,X))** succeed, but **g(X,Y,Z)** fails. (1 backtrack)

16. **Z=2** is tried and fails **g(X,Y,Z)**, followed by **Z=3**, **Z=4**, **Z=5**. So **n(Z)** fails. (5 backtracks)

17. **Y=2** is tried and fails **g(X,Y,Z)** after all **Z** are tried. (6 backtracks)

18. **Y=3** is tried. **Z=1**, **Z=2**, **Z=3**, and **Z=4** are tried and fail; **Z=5** succeeds. (4 backtracks)

(d) First create possibility lists for **X**, **Y**, and **Z**, using the **n** and **f** predicate expressions:

```
X: [1,3,4,5]
Y: [1,2,3,4,5]
Z: [1,2,3,4,5]
```

The only multivariable constraints are **X>Y** and **g(X,Y,Z)**.

1. Pick variable **X** first since it has the fewest possibilities. **X=1** is not possible with **X>Y**. And **X=5** is not possible with **g(X,Y,Z)**. So **X=3** or **X=4** only. (Note it's not fair to say that **X** cannot be 3 now because that could only be shown by considering the two constraints together, **X>Y** and **g(X,Y,Z)**. Pure relaxation throws away bindings from one constraint when considering another constraint.)

2. Pick variable **Y**. **Y=3** is the only way to satisfy constraint **g(X,Y,Z)**.

3. Return to variable **X**. Now by **X>Y**, **X** cannot be 3, so it must be 4.

4. Go to variable **Z**, the only remaining variable with multiple possibilities. It must be 5 by **g(X,Y,Z)**.

13-14. (a) Whenever a unique value is chosen for a variable the **unique_values** predicate can exploit that new value. So **unique_values** becomes harder to satisfy, and it must do more backtracking. This slows the program down. When we start with two values for e, things aren't slowed down until o is found to be 0, at which point there are fewer other unassigned variables to worry about, so the slowdown is less dramatic.

(b) Delete the definition of **unique_values**, and delete its call from the definition of the **satisfiable** predicate. It isn't necessary.

(c) The predicate **duplication** catches most of the bad assignments of variables to values, but **unique_values** catches a few cases that **duplication** misses. For instance, if each of variables A, B, and C have possibilities of 4 or 5, and each variable has a distinct value, there's no way that the conditions can be met; **unique_values** will catch this.

13-18.(a) A state is any set of possibility lists for the variables, where possibility lists are subsets of the original possibility lists in the starting state, plus a list of which variables are "active". It can be represented as a list of lists.

(b) Depth-first search. It keeps moving ahead, never returning to a previous state. There's no point in returning to a previous state because you keep learning useful things as you proceed, things that can potentially help in future study.

(c) No, because you can't predict what intermediate states you'll get into.

(d) The branching factor always stays the same or decreases by 1. A state transition means either elimination of a single possibility for a variable or no elimination.

(e) If the sum of the number of possibilities on all initial possibility lists is N, then there are |2 sup N| possibility list configurations obtainable by crossing out 0 to N possibilities. But we haven't included the active variables. There are |2 sup V| configurations of these for each possibility-list configuration, so the total number of states is the product of these two numbers, or |2 sup {N + V}|. (Some of those states have zero possibilities for some variables, but such states are possible when the input has errors.)

14-1. **X** binds to 5, and **Y** binds to **X**, so **Y** is bound to 5 too. So the **possible**s cancel, giving as resolvent

```
state(3,A,5); state(3,6,5).
```

But the second predicate expression is covered by the first, so we can rewrite this as

```
state(3,A,5).
```

That is the only resolution possible.

14-2. (a) Suppose:

t represents your having travel money;
d represents your department chairman approval;
s represents sponsor approval;
b represents the boss of your department chairman approving;
f represents discretionary funds being available.

Then we can write the initial rules as

```
t :- d, s.
t :- d, b, f.
```

(b)

```
t; not(d); not(s).
t; not(d); not(b); not(f).
```

(c)

```
(d,s);(d,b,f) :- t.
```

or equivalently:

```
d,(s;(b,f)) :- t.
```

(d) The second preceding can be translated to

```
(d, (s;(b,f))); not(t).
```

Using the distributive law of "or" over "and":

```
(d; not(t)), (s; (b,f); not(t)).
```

```
d; not(t).
s; (b,f); not(t).
```

The first of those two is a clause, but we still need to use the distributive law of "or" over "and" on the second. Finally, we have these three clauses:

```
d; not(t).
s; b; not(t).
s; f; not(t).
```

(e) Just resolve the fact **t** with the clauses in part (d), giving:

```
d.
s; b.
s; f.
```

Since we did all possible resolutions involving all given information, those are all the clauses derivable.

(f) Just resolve the clause **not(t)** with the clauses in part (b), giving:

```
not(d); not(s).
not(d); not(b); not(f).
```

14-5. Resolvent clauses are only guaranteed to *include* the truth, not give a precise specification of the truth as the "and" of the two resolvents would. The statement "b or c" does include all the cross-hatched region in the figure, the "and" of the input clauses, which is all that is required.

14-9. (a) The number of possible pairs of clauses or |N (N - 1) / 2|. (Note that resolution is a commutative operation, hence the division by 2.)

(b) Each successful resolution increases by one the number of clauses, so after K resolutions there will |N + K| clauses and |(N + K) (N + K - 1) / 2| pairs. K of these pairs will have already been used, so an upper bound on the branching factor is |((N + K) (N + K - 1) / 2) - K |.

15-1. The real test is not the opinion of any human being, but whether the car works correctly when the supposed problem is fixed. So the gold standard is the minimum change necessary to get the car working right again. (It's the "minimum" because you can always get any faulty car working again by simply replacing every part with a nonfaulty part.)

15-3. In the confusion matrix, examine the cell pairs symmetric about the diagonal running northwest-southeast (but not including the cells of the diagonal itself.) Look for any pair where (1) one number is much larger than the other, and (2) the cell with the larger number is in a row corresponding to diagnosis rules that precede the rules that correspond to the row for the other diagnosis; such a pair suggests the order of the rows should be reversed. This clue is further strengthened if the other cells in the same two rows and two columns are small, except for the diagonal terms.

15-6. (a) Not a good idea by the third and fourth overall criteria of Section 15.10.

(b) Not a good idea by the first, second and sixth criteria.

(c) Not a good idea by the second and third criteria.

(d) Not a good idea by the third, fourth, and fifth criteria.

M-1. (a) Yes, backtracking on failure is the whole point.

(b) No, the whole point of forward chaining is to keep moving forward from facts to new conclusions. Choosing new facts and new rules is better done by iteration, not backtracking.

(c) No, since you store things on an agenda and just pick the best thing on the agenda at each point, a form of iteration. The agenda item selected might be something in a far away and/or deeper part of the search lattice than the last agenda item, so you're not necessarily "going back" to anything.

(d) Yes, dependency-based backtracking is included in this.

(e) Yes, because you might have several directions (links) you could follow to find a value for some slot. If one path doesn't pan out, you should backtrack and try another.

(f) No, because it's just an example of forward chaining.

M-3. Horn Clause: f. (the definition)

generate-and-test: i. (variable **X** is generated, then tested)

constraint: m. (equivalent terminology)

forward chaining: o. (that's the whole idea) default: b. (the definition)

depth-first control structure: q. (essential even for those depth-first activities that aren't searches)

intension: p. (the definition)

difference table: h. (the main purpose)

near miss: n.

Skolem Function: j.

caching: a. (the definition)

dependency: r. (the definition for predicate expressions in the same query)

multiple inheritance: c. (the chief difficulty)

decision lattice: g.

Bayes' Rule: d. (it can be proved from probability theory)

heuristic: e. (a famous example)

hierarchical reasoning: l. (hierarchy is in the levels of recursion)

agenda: k. (the definition)

M-4.(a) 1, and-or-not lattice. This sounds like a pretty straightforward application for expert systems; few variables will be needed. So the lattice will be feasible, and will be a simple (and reasonably efficient) implementation method. You could build the lattice with logic gates on a VLSI chip that could be put into a small hand-held device. Dependency-based backtracking is probably too fancy a technique for this problem; you must wait for the user to answer questions, so the extra speed of dependency-based backtracking will usually be wasted. You shouldn't need to backtrack anyway since this sounds good for forward chaining; there are few facts (the problem mentions "simple questions"), and probably low "fanout" from facts (multiple references to the same fact in different rules should be rare). A breadth-first resolution strategy is a very slow method of reasoning only appropriate when you want to be sure to reach every valid conclusion, including conclusions with "or"s; here we only want a single conclusion about which tax form to use. Finally, there's no such thing as "confusion-matrix conflict resolution"; conflict resolution is how you decide what to do next in a rule-based system, and confusion matrices are statistical summaries useful in analyzing experiments with system performance.

(b) 4, concurrency. You can only ask questions one at a time, and there's probably low fanout of facts to rules, so there isn't much opportunity for concurrency. (Things are quite different for expert systems with real-time inputs, like the "smart house" in Exercise 6-12.) Virtual facts would be essential if you used backward chaining--you don't want to ask the same question twice. Inference is a general term for all reasoning, and this system must reason. A depth-first control structure is good for many expert systems including this one, regardless of whether they do backward, forward, or hybrid chaining, because it's easy to implement.

(c) 2, **repeat**. There's not much reason to repeat anything here; after all, an and-or-not lattice sound good, and that can't accommodate iteration very well. The **asserta** or **assertz** would be necessary for virtual facts. The **write** would be necessary to ask questions of the user. The > could be used in making the needed monetary-amount comparisons.

M-5. (a) 1, best-first search. Deviation from a straight line is a good (and necessary) evaluation function; we clearly don't want to lurch a lot, or we couldn't walk very fast. But there's no obvious cost function; speed is not important, according to the problem statement. And there's no such thing as "weight-first search".

(b) 2, caching only. Caching is helpful for reexamining the same ground as robot moves forward. Means-ends analysis isn't helpful because there's only one operator, foot placement. So all the tricks that means-ends analysis uses to select operators based on logical conditions don't have anything to work with. Furthermore, the problem is quantitative: it depends on numeric things like the coordinates of a square, the positions of the other legs, the center of gravity of the robot, the direction of motion of the robot, and so on. Means-ends analysis is intended for non-quantitative problems.

(c) 2, **assertz**. No reason to use queues here, about the only reason to ever use an **assertz**. The **is** is always useful with numeric calculations, and the cut and **repeat** predicates could help make search efficient, just as they did for the best-first program in Section 10.9.

Go to book index

```
delim %%


predicate expression




predicate               arguments
(alias predicate
name)




ghwizhge(ggirjajdj,365,ajgc52)




first                   second                  third
argument                argument                        argument
```

Figure 2-1: terminology about predicate expressions

```
|type|property|relationship|database|function|probability
|predicate|predicate|predicate|predicate|predicate|predicate

Number|1|2|2|1 or more|2 or more|1 or more
of
arguments
Nature|a thing|a thing and|two things|a thing|last is|last is
of||a property||and|result of|probability
arguments||||properties|operation|of the
|||||on others|fact truth
Description|gives a class|gives|describes|like a|describes|variant of
|the thing|property of|relationship|data record|a function|preceding
|belongs to|a thing|of two||mapping|for partly
|||things|||certain facts
Examples|ship|color|part_of|ship|sum|color
|(kennedy)|(kennedy,|(kennedy,|(kennedy,|(3,|(kennedy,
||gray)|u_s_navy)|gray,|4,|gray,
|vehicle|||14n35e,|7)|0.8)
|(ship)|location|a_kind_of|021685)
||(kennedy,|(kennedy,
||14n35e)|ship)|
Meaning|"The Kennedy|"The color of|"The Kennedy|"There is|"The sum|"We believe
of the|is a ship"|the Kennedy|is part of the|a ship|of 3 and|with certainty
examples||is gray"|U.S. Navy"|record with|4 is 7"|0.8 that the
|"A ship is|||entries||Kennedy
|a vehicle"|"The location|"The Kennedy|Kennedy,||is gray"
```

```
||of the Kennedy|is a kind|gray, and
||is 14n35e"|of ship"|14n35e"
```

Figure 2-2: summary of the kinds of predicates in Chapter 2

```
gray                                    government

        color                                   a_kind_of

ship                                    u_s_government

                                                            has
        a_kind_of                               part_of
                a_kind_of

enterprise              kennedy                 u_s_navy

                                part_of                 civil_service_system
                part_of

        location

15n35e
```

Figure 2-3: an example semantic network

```
Symbol|Description|Meaning
?-|question mark,|query prompt
|minus sign|(typed by Prolog)
.|period|(1) query terminator (typed by user)
||(2) database fact and rule terminator
, |comma|(1) "and" of predicate expressions in a query
||(2) separator of arguments
||(3) separator of list items (see chapter 5)
;|semicolon|(1) "or" of predicate expressions in a query
||(2) forces new alternative after a query answer
()|parentheses|(1) for grouping arguments together
||(2) for grouping query expressions
_ |underscore|no special meaning, but we often use
||as a character to make long names legible
:- |colon,|rule definition symbol (see
|minus sign|chapter 4); read as "if"
[] |square brackets|a list (see chapter 5)
<> |angular brackets|not a Prolog symbol, but
||we use it to enclose descriptions of
||things (like Backus-Naur Form)
```

Figure 3-2: special symbols used with Prolog in this book

Query:

```
?-boss(X,Y), boss(Y,Z).
```

Database:

```
boss(dick,harry).
boss(tom,dick).
boss(ann,mary).
boss(mary,harry).
```

Processing:

```
Step number|First boss predicate expression:|Second boss predicate expression:
in text|Bindings made|Binding of Z
|of X and Y

1|X=dick,Y=harry|
2||[no Z possible]
3|X=tom,Y=dick|
4||Z=harry
5||
```

Figure 3-3: processing state at generation of first answer to superboss query: X=tom,Y=dick,Z=harry

---

Query:

```
?-boss(X,Y), boss(Y,Z).
```

Database:

```
boss(dick,harry).
boss(tom,dick).
boss(ann,mary).
boss(mary,harry).
```

Processing:

```
Step number|First boss predicate expression:|Second boss predicate expression:
in text|Bindings made|Binding of Z
|of X and Y
1|X=dick,Y=harry|
2||[no Z possible]
3|X=tom,Y=dick|
4||Z=harry
5||
6||
7||[no further Z possible]
8|X=ann,Y=mary|
9||Z=harry|
10||
```

Figure 3-4: processing state at generation of second answer to superboss query: X=ann,Y=mary,Z=harry

---

Query:

```
?-boss(X,Y), not(boss(X,dick)), boss(Y,Z).
```

Database:

```
boss(dick,harry).
boss(tom,dick).
boss(ann,mary).
boss(mary,harry).
```

Processing:

```
Step number|First predicate|Second predicate|Third predicate
in text|expression:|expression:|expression:
|Bindings made|What happens?|Binding of Z
|of X and Y
1|X=dick,Y=harry
2||succeeds
3|||[no possible Z]
4||fails
4|X=tom,Y=dick
5||fails|
6|X=ann,Y=mary
7||succeeds
8|||Z=harry
```

Figure 3-5: processing order for the extended superboss example: answer X=ann,Y=mary,Z=harry

Database:

Memory location|Contents

10000|a_kind_of(enterprise,ship). 10026|a_kind_of(kennedy,ship). 10052|part_of(enterprise,u_s_navy). 10070|part_of(kennedy,u_s_navy). 10088|part_of(u_s_navy,u_s_government).

Index:

Predicate name|Locations of facts using it

a_kind_of|10000,10026 part_of|10052,10070,10088 Figure 3-6: an example of predicate indexing

```
left side                        special symbol          right side
(a predicate             meaning "if"            (a query)
expression)


a(X,middle,Y) :- b(X), c(X,Y,extra), d(X,Z), e(Z,Y), f(Z).


parameter variables              local variable          rule terminator
```

Read a rule as meaning: the left side is true if querying the right side succeeds. Figure 4-1: terminology about rules

Query:

```
?-boss(X,Y), not(boss(X,tom)).
```

Database:

```
department(tom,sales).
department(harry,production).
manager(dick,sales).
manager(mary,production).
boss(B,E) :- department(E,D), manager(B,D).
```

Processing:

```
Step|In rule|In rule|In query:|In rule|In rule|In query:
in text|call 1:|call 1:|Bindings|call 2:|call 2:|argument
|Bindings|Binding|to X and Y|Bindings|Binding|to not
|to E and D|to B||to B and E|to D|

1|E=tom,
|D=sales
2||B=dick|
2|||X=dick,
|||Y=tom
3||||
4||||B=dick,
||||E=tom
4|||||D=sales
4||||||succeeds
5||[no more
||choices]
6|E=harry,
|D=production
7||B=mary
7|||X=mary,
|||Y=harry
8||||B=mary,
||||E=tom
9|||||D=sales
10||||||fails
```

Figure 4-2: processing order of the other-than-tom's-boss query: answer X=mary,Y=harry

---

```
X       r       Z       r       Y


                r
```

Figure 4-3: the general form of transitivity

```
                d


                c


                b


                a
```

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

Figure 4-4: a transitivity example

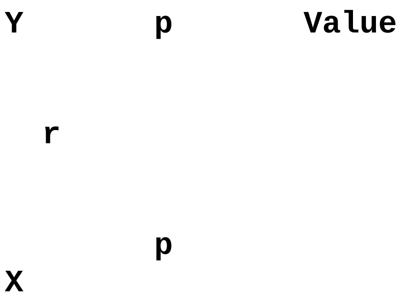


Figure 4-5: the general form of inheritance

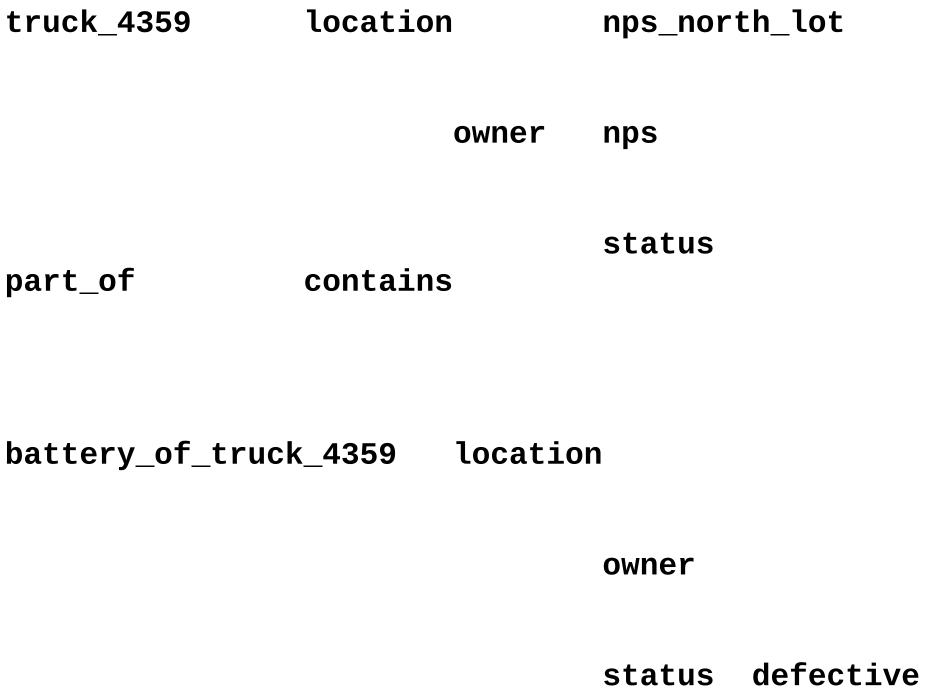
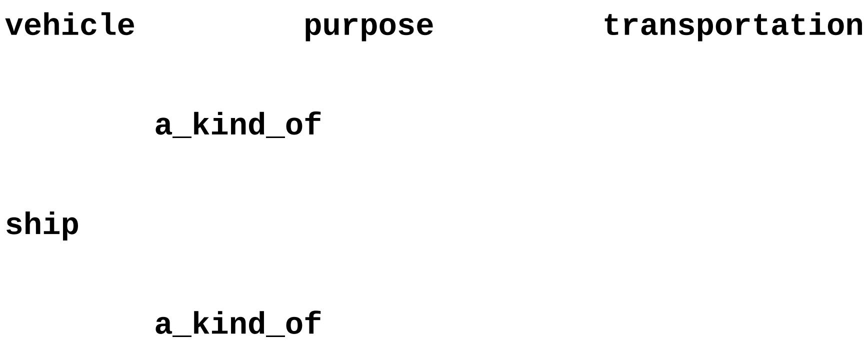


Figure 4-6: an inheritance example



```
                                a_kind_of
carrier


        a_kind_of


vinson                          purpose
```

Figure 4-7: an example of both transitivity and inheritance

```
                action




                                        pedsignals


safe_stop_possible   pedhalt   pedgo            greenfacing




                light                           clockwise_cross
```

Figure 4-8: the predicate hierarchy for the traffic lights program

---

```
 D
 C      G
 B      F   I
 A      E   H
---------------------------
```

Figure 4-9: some blocks on a table

Predicate|*Arguments*|*What it means*

first(L,I)|a list,|I is the first |an item|item of the list L last(L,I)|a list,|I is the last |an item|item of the list L member(I,L)|an item,|I is an item somewhere |a list|inside the list L length(L,N)|a list,|N is the length |a number|of the list L max(L,N)|a list,|N is the largest number |a number|in the list of numbers L delete(I,L1,L2)|an item,|L2 is L1 with all |a list,|occurrences of the I |a list|removed from it append(L1,L2,L3)|a list,|L3 is the glueing |a list,|together of the |a list|lists L1 and L2 subset(L1,L2)|a list,|all items in L1 |a list,|are in L2 sort(L1,L2)|a list,|L2 is the sorting |a list|of list of numbers L1 Figure 5-1: the classic list-processing predicates defined in this chapter (to be used in the rest of this book)

Given the definition of append:

```
append([],L,L).
append([X|L],L2,[X|L3]) :- append(L,L2,L3).
```

Given the query:

```
?- append([gas,oil],[tires,battery,radiator],Things_to_check_before_trip).
```

Nested environments created:

```
Query: ?- append([gas,oil],[tires,battery,radiator],Things_to_check_before_trip).
In definition line 2: [X|L]=[gas,oil], L2=[tires,battery,radiator], [X|L3]=???
So X=gas, L=[oil], [X|L3]=[gas|???]


        Query: ?- append([oil],[tires,battery,radiator],L3).
        In definition line 2: [X|L]=[oil], L2=[tires,battery,radiator], [X|L3]=???
        So X=oil, L=[], [X|L3]=[oil|L3]


                Query: ?- append([],[tires,battery,radiator],L3).
                In definition line 1: L=[tires,battery,radiator]
                Query succeeds
```

Note: "???" means unknown Figure 5-2: an example of using append with first two arguments bound (inputs) and last argument unbound (an output); the processing state is just after the innermost recursion succeeds

---

Given the definition of append:

```
append([],L,L).
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).
```

```
first|second|third|What happens?|Example
arg.|arg.|arg.|
```

bound|bound|bound|Answers yes if third|?- append([a,b],[c],[a,b,c]). |||argument is the first two|yes. |||arguments glued together |||(in order), else no bound|bound|unbound|Glues the second argument|?- append([a,b],[c,d],L). |||on the end of the first|L = [a,b,c,d] |||argument, binds result |||to the third argument bound|unbound|bound|Checks if the first|?- append([a,b],L,[a,b,c]). |||argument is on the,|L = [c] |||front of the third,| |||binding the second to |||the rest of the third unbound|bound|bound|Checks if the second|?- append(L,[c,d],[a,b,c,d]). |||argument is on the|L = [a,b] |||back of the third, |||binding the first to |||the rest of the third unbound|unbound|bound|Generates all divisions|?- append(L1,L2,[a,b,c]). |||of the third argument|L1 = [], L2 = [a,b,c]; |||into two pieces|L1 = [a], L2 = [b,c]; |||(preserving term order)|L1 = [a,b], L2 = [c]; ||||L1 = [a,b,c], L2 = [] unbound|bound|unbound|Generates in abstract|?- append(L1,[a,b],L2). |||form all results of|L1 = [], L2 = [a,b]; |||glueing something on|L1 = [_1], L2 = [_1,a,b]; |||the front

of the|L1 = [_1,_2], L2 = [_1,_2,a,b] |||second argument|(The _1 and _2 ||||are Prolog-invented ||||variable names.) bound|unbound|unbound|Generates in abstract|?- append([a,b],L1,L2). |||form all results of|L1 = _1, L2 = [a,b|_1] |||glueing something on|(The _1 is a |||the back of the|Prolog-invented |||first argument|variable name.)

Note: any bound arguments to append must be lists for the above program to work.

Figure 5-3: the different possible uses of the append predicate

---

increasing
flexibility

meta-rules

backwards forwards hybrid chaining chaining chaining virtual facts

partitioning

parallelism and-or-not lattices decision lattices caching

increasing
efficiency

Figure 6-1: the spectrum of control-structure concepts for rule-based systems

---

Step number|Which fact|Which rule|What happens?
|being pursued?|matched?

***1|fact2*|R8|Define R10: d :- fact3. |fact2|R9|Define R11: e :- fact4. 2|fact3|R10|New fact found: d. 3|d|R4|nothing |d|R5|New fact found: b. 4|b|R2|Define R12: goal1 :- a. 5|--|-- 6|not(d)|R4|nothing, since d a fact |not(e)|R7|New fact found: c. 7|c(2)|R3|New fact found: goal2(2).**

Figure 6-2: summary of the example of pure forwards chaining

Pure backwards chaining:|Pure forwards chaining:|Rule-cycle hybrid:

query order|fact order|rule order

has priority over|has priority over|has priority over

rule order|rule order|rule-right-side order

which has priority over|which has priority over|which has priority over

rule-right-side order|rule-right-side order|fact order

which has priority over

fact order

Figure 6-3: ordering priorities for three kinds of chaining

Cycle no.	*R1*	*R2*	*R3*	*R4*	*R5*	*R6*	*R7*	*R8*	*R9*
1	**fails**	**fails**	**fails**	**skipped**	**fails**	**fails**	**skipped**	**succeeds**	**fails**
2	**fails**	**fails**	**fails**	**skipped**	**succeeds**	**--**	**skipped**	**--**	**fails**
3	**fails**	**fails**	**fails**	**skipped**	**--**	**--**	**skipped**	**--**	**fails**
4	**fails**	**fails**	**fails**	**fails**	**--**	**--**	**succeeds**	**--**	**fails**
5	**fails**	**fails**	**succeeds**	**--**	**--**	**--**	**--**	**--**	**--**

Figure 6-4: summary of the example of rule-cycle hybrid chaining (the "--" means not necessary to test)

```
                        appliance works?

                no                      yes

        plugged in?                     what kind of appliance?

        no      yes                             heating         mechanical      other

plug it in      are nearby electrical           [further        [further        [further
and stop        appliances working?             questions]      questions]      questions]

                no              unclear         yes

                        plug into different
                        outlet; work now?

                no                      yes

blown fuse or                           stop
circuit breaker?

yes             no
```

[must be internal problem,
so ask more about this]

fix fuse or
circuit breaker
and stop

Figure 6-5: a decision lattice

---

and parallelism

partition parallelism

```
a(X) :- b, c(X), d(X,Y).
```

```
r(X) :- s(X), t(X,Y).
r(X) :- u(X).
```

or parallelism

```
a(X,Y) :- e(X,Y), f(X,Y), not(c(X)).
```

variable-matching parallelism

Figure 6-6: types of parallelism

---

question

noun_phrase linker modifier ?

modifier noun prepositional_phrase linker modifier ?

which noun prepositional_phrase linker modifier ?

which ships prepositional_phrase linker modifier ?

which ships preposition noun_phrase linker modifier ?

```
which ships in noun_phrase linker modifier ?

which ships in the noun linker modifier ?

which ships in the Mediterranean linker modifier ?

which ships in the Mediterranean are modifier ?

which ships in the Mediterranean are American ?
```

Figure 6-8: a parsing example (downwards arrows represent top-down parsing, upwards arrows represent bottom-up)

```
                diagnosis



power_problem     klutz_user   mechanical_problem   heating_element



                                askifnot

                        askif

        ask   affirmative   askif2

        questioncode
```

Figure 7-2: the predicate hierarchy for the appliance diagnosis expert system, including the problem-independent predicates

```
                                        forwards

done                    fact                    pursuit
```

```
[user-defined]  [user-defined]


                        rule                    rule_pursuit
                        [user-defined]


                        member          delete          new_rule
```

Figure 7-3: the predicate hierarchy for the forwards-chaining program

```
                        n1


                a?                 not(a)?


                n2                      n3


        d?              not(d)?  c?             not(c)?


        n4              u               n5              n6


e?              not(e)?         b?    not(b)?   d?      not(d)?


u               r                       t       v               t               s
```

Figure 7-4: a derived decision tree

```
Independence-assumption "or":
        p(A or B) = p(A) + p(B) - p(A)p(B)
Independence-assumption "and":
        p(A and B) = p(A)p(B)
```

```
Conservative "or":
        p(A or B) = p(A) if p(A)>p(B), else p(B)
Liberal "and":
        p(A and B) = p(B) if p(A)>p(B), else p(A)
```

```
Liberal "or":
        p(A or B) = p(A)+p(B) if this < 1, else 1
Conservative "and":
        p(A and B) = p(A)+p(B)-1 if this > 0, else 0
```

p(A)

p(B)

Figure 8-2: Venn diagrams illustrating the three standard probability-combination methods, applied to two probabilities

---

|and-combination|or-combination

Independence assumption|%p sub 1 p sub 2 ... p sub n%|%1 - [(1 - p sub 1) (1 - p sub 2) ... (1 - p sub n)]% (reasonable-guess) Conservative assumption|%max (0 , (p sub 1 + p sub 2 + ... + p sub n) - n + 1)%|%max (p sub 1 , p sub 2 , ... p sub n)% (lower bound) Liberal assumption|%min (p sub 1 , p sub 2 , ... p sub n)%|%min (1 , p sub 1 + p sub 2 + ... + p sub n) % (upper bound)

Note:

> **conservative "and" % <= % independence-assumption "and" % <= % liberal "and"**
> **% <= % conservative "or" % <= % independence-assumption "or" % <= % liberal "or"**

Figure 8-3: formulas for combination of probabilities %p sub i%, i=1 to n

---

	definitely	probably	probably not	definitely not
definitely	definitely	definitely	definitely	definitely
probably	definitely	probably	probably	probably
probably not	definitely	probably	probably not	probably not
definitely not	definitely	probably	probably not	definitely not

Figure 8-5: one evidence combination method

---

	definitely	probably	probably not	definitely not
definitely	definitely	definitely	definitely	definitely
probably	definitely	definitely	definitely	probably
probably not	definitely	definitely	probably	probably not
definitely not	definitely	probably	probably not	definitely not

Figure 8-6: another evidence combination method

```
        b       c
a                       e       f
                d
                        g       h
```

Figure 9-2: the equivalent search graph for Figure 9-1

```
                car assembled
                except for screws
                on battery removed
  remove                                         remove cables
  screws
                                        car assembled
                                        except for screws
car assembled                   and cables on
                                        the battery
  remove
  cables                                                        remove battery


                car assembled           remove          car assembled
                except for cables               screws          except battery
                on battery removed                              removed
```

Figure 9-3: part of a search graph for auto repair

```
a :- b,c.                               a :- c.         pursue d
c :- d.         pursue b        c :- d.
b.                                      d.                      a :- c.
d.                                                              c.
pursued: none                   pursued: b              pursued: b,d


        pursue d                        pursue b                        pursue c
                        a :- b,c.
a.
                        b.              pursue c
pursued: d,b,c
                        c.
                        pursued: d
                                                a :- b.         pursue b
                                                b.
```

```
                                                  pursued: d,c
```

Figure 9-4: a complete search graph for generalized (any-fact-order) forwards chaining from a particular set of facts and rules (rules no longer useful are deleted from states as search proceeds)

```
Name of|Uses|Uses|Uses|Next state whose
search strategy|agenda?|evaluation|cost|successors are found
||function?|function?
```

Depth-first|no|no|no|A successor of the last search||||state, else the predecessor Breadth-first|yes|no|no|The state on the search||||agenda the longest Hill-climbing|no|yes|no|The lowest-evaluation (optimization)||||successor of the last state Best-first|yes|yes|no|The state on the agenda search||||of lowest evaluation value Branch-and-bound|yes|no|yes|The state on the agenda ||||of lowest total cost A* search|yes|yes|yes|The state on the agenda ||||of lowest sum of evaluation ||||value and total cost

Figure 9-5: the classic search strategies (heuristics may be used with any of these)

Given the rule set:

```
t :- a, b.
t :- c.
u :- not(a), c.
u :- a, d.
v :- b, e.
```

Then this is a decision lattice implementing it:

```
                         a?


        yes                     no


        b?                      c?


yes             no      yes             no


conclude        c?      conclude                b?
t                       t


        yes     no              yes             no
```

```
conclude        d?                      e?              fail
t
        yes             no              yes     no


        conclude        fail            conclude        fail
        u                               v
```

Figure 9-10: the decision lattice for backwards chaining on an example rule set

---

Assume the rules:

```
t :- a, b.
t :- c.
u :- not(a), c.
u :- a, d.
v :- b, e.
```

And assume the only possible facts are a, b, c, d, and e, provided for a database in order of priority a, c, b, d, and e. Assume each has an independent probability of occurrence P. Situations to consider:

a?	c?	b?	d?	e?	probability	number of	conclusion
						matches	reached
false	true	-	-	-	%P (1 - P)%	1	t
true	true	-	-	-	%P sup 2%	3	t
true	false	true	-	-	%P sup 2 (1 - P)%	3	t
true	false	false	true	-	%P sup 2 (1 - P) sup 2%	3	u
true	false	false	false	true	%P sup 2 (1 - P) sup 3%	4	-
true	false	false	false	false	%P sup (1 - P) sup 4%	3	-
false	false	true	true	true	%P sup 3 (1 - P) sup 2%	4	v
false	false	true	true	false	%P sup 2 (1 - P) sup 3%	4	-
false	false	true	false	true	%P sup 2 (1 - P) sup 3%	3	v
false	false	true	false	false	%P (1 - P) sup 4%	3	-
false	false	false	true	true	%P sup 2 (1 - P) sup 3 %	3	-
false	false	false	true	false	%P (1 - P) sup 4%	2	-
false	false	false	false	true	%P (1 - P) sup 4%	2	
false	false	false	false	false	%(1 - P) sup 5%	1	-

Figure 9-11: analysis of cost of forwards chaining on the rule set

---

Given this array of edgeness measures for a picture (array g(i,j)):

0	1	0	8	2
1	2	7	3	1
3	1	5	2	0
2	2	3	3	1

```
2|5|2|1|3
1|7|1|2|1
3|6|2|2|1
```

Inference: there's a continuous edge running north-northeast to south-southwest, though it's hard to see towards the center of the picture. That's the interpretation (array e(i,j)):

```
false|false|false|true|false
false|false|true|false|false
false|false|true|false|false
false|false|true|false|false
false|true|false|false|false
false|true|false|false|false
false|true|false|false|false
```

Figure 9-13: example arrays for finding visual edges

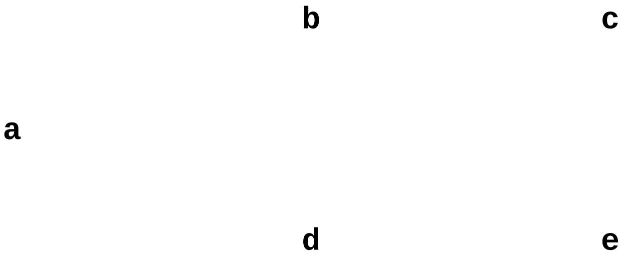


Figure 10-3: a simpler route-planning problem

Step number|depthsearch2|depthsearch2|depthsearch2|depthsearch2
in text|first call|second call|third call|fourth call

1|called with |state a 2|first rule |fails 3|second|called with |rule tried|state b 4||first rule|called with ||fails,|state c ||second tried 5|||both rules |||fail on c 6||backtrack to ||not, then to ||successor; ||choose d 7|||called with |||state d; |||first rule |||fails; 2nd |||rule picks |||b, which |||fails not 8|||choose e|called with ||||state e 9||||first rule ||||succeeds ||||with path ||||list [e,d,b,a] |||succeeds ||succeeds |succeeds

Figure 10-4: summary of the depth-first search example

breadthsearch
find_successors
cleandatabase
measurework

length

successor *[user-written]*

goalreached *[user-written]*

Figure 10-5: the rule-predicate hierarchy for the breadth-first search (breadthsearch) **program**

---

bestsearch

pick_best_state **add_successors** **cleandatabase** **repeatifagenda** **measurework**

special_less_than **add_state** **checkabolish**

countup

eval *[user-written]*

successor *[user-written]*

goalreached *[user-written]*

Figure 10-6: the rule-predicate hierarchy for the best-first search (bestsearch) **program**

---

astarsearch

pick_best_state **add_successors** **cleandatabase** **repeatifagenda** **measurework**

special_less_than **add_state** **checkabolish**

countup

agenda_check **usedstate_check**

fix_agenda

replace_front

append

cost *[user-written]* **eval** *[user-written]* **successor** *[user-written]* **goalreached** *[user-written]*

Figure 10-7: the rule-predicate hierarchy for the A* search (astarsearch) program

---

Desired|Should|Should|Should|Should|Should
situation|you|you|you|you|you
|visit|visit|visit|visit|visit
|cafeteria?|vending|your|secretary?|colleague?
||machine?|office?

You want|yes|yes|no|no|no to be not| hungry, and you're hungry now You want|no|no|yes|yes|no to have change, and you don't now You want|no|no|yes|yes|yes to know| where something is Figure 11-1: tabular representation of the operator recommendations (difference table) for the office-worker problem

---

Desired|Should|Should|Should|Should|Should
situation|you|you|you|you|you
|visit|visit|visit|visit|visit
|vending|cafeteria?|colleague?|secretary?|your|
|machine?||||office?

You want|yes|yes|no|no|no to be not| hungry, and you're hungry now You want|no|no|no|yes|yes to have change, and you don't now You want|no|no|yes|yes|yes to know||||(but try|(but try where||||third)|second) something is Figure 11-2: a different tabular representation of the operator recommendations (difference table) for the office-worker problem

---

Desired|Should|Should|Should|Should|Should|Should|Should|Should
situation|you|you|you|you|you|you|you|you
|replace|replace|disassemble|disassemble|assemble|assemble|turn over|smash
|batteries|light|case|top|case|top|case|case
|?|?|?|?|?|?|?|?

You want|yes|no|no|no|no|no|no|no batteries OK, when they aren't You want|no|yes|no|no|no|no|no|no light OK, when it isn't You want|no|no|yes|no|no|no|no|yes case open, when it isn't You want|no|no|no|yes|no|no|no|yes top open, when it isn't You want|no|no|no|no|yes|no|no|no case closed, when it isn't You want|no|no|no|no|no|yes|no|no top closed, when it

isn't You want|no|no|no|no|no|no|yes|yes batteries outside, when they aren't

Figure 11-4: tabular representation of the operator recommendations (difference table) for the flashlight problem

---

replace_batteries

precondition
postcondition
satisfaction
satisfaction

postcondition
satisfaction

disassemble_case
assemble_case

turn_over_case

Figure 11-6: the form of the solution of the dead-batteries flashlight problem by means-ends analysis (Figure 11-7 gives a more detailed picture)

.DS L ***Level 1 State=[closed(case),closed(top),inside(batteries),defective(batteries), ok(light),unbroken(case)] Goal=[ok(batteries),closed(case),closed(top)] Oplist=[disassemble_case,turn_over_case,replace_batteries,assemble_case] Goalstate=[closed(case),inside(batteries),ok(batteries),closed(top), ok(light),unbroken(case)] Operator=replace_batteries Level 2 (Precondition recursion) (Postcondition recursion) State=[closed(case),closed(top), State=[inside(batteries),ok(batteries), inside(batteries),defective(batteries), open(case),closed(top), ok(light),unbroken(case)] ok(light),unbroken(case)] Goal=[open(case),outside(batteries),] Goal=[ok(batteries),closed(case), unbroken(case)] closed(top)] Oplist=[disassemble_case,turn_over_case] Oplist=[assemble_case] Goalstate=[outside(batteries),open(case), Goalstate=[closed(case),inside(batteries), closed(top),defective(batteries), ok(batteries),closed(top), ok(light),unbroken(case)] ok(light),unbroken(case)] Operator=disassemble_case Operator=assemble_case Level 3***

(Precondition (Postcondition (Precondition (Postcondition recursion) recursion) recursion) recursion) [nothing State=[open(case),closed(top), [nothing [nothing needed] inside(batteries),defective(batteries) needed] needed] ok(light),unbroken(case)] Goal=[outside(batteries)] Oplist=[turn_over_case] Goalstate=[outside(batteries),open(case), closed(top),defective(batteries), ok(light),unbroken(case)] Operator=turn_over_case Level 4 (Precondition recursion) (Postcondition recursion) [nothing needed] [nothing needed]

Figure 11-7: the recursive calls in the dead-batteries flashlight problem, with bindings eventually found for Oplist, Goalstate, and Operator

---

```
                        frame_name: business_form
                        a_kind_of: physical_object
                        type:
                        author:


                        frame_name: memo
                        a_kind_of: business_form
                        type: informal
                        author:
                        addressees:
                        subject:
                        date:
                        text:


frame_name: memos of Ann                frame_name: memos to Ann
frame_name: budget memos
a_kind_of: memo                 a_kind_of: memo                 a_kind_of: memo
type: informal                          type: informal                          type:
informal
author: Ann (D)                         author:
author:
addressees:                                     addressees: [Ann] (D)
addressees:
subject:                                        subject:
subject: budget (D)
date:                                           date:
date:
text:                                           text:
text:




                                        frame_name: budget memos to Ann from Tom
                                        a_kind_of: memos to Ann
                                        a_kind_of: budget memos
                                        type: informal
                                        author: Tom
                                        addressees: [Ann] (D)
                                        subject: budget (D)
```

```
date:
text:


frame_name: budget memo to Ann from Tom on
6/12

a_kind_of: budget memos to Ann from Tom
type: informal
author: Tom (D)
addressees: [Ann] (D)
subject: budget (D)
date: 6/12 (D)
text: "Error in personnel estimate:
        10, not 10,000."
```

Notes: each box is a frame, and arrows represent a_kind_of facts. Slot names are given, then a colon, and then the slot value; a "(D)" means the value follows by the definition of the frame.

Figure 12-1: some frames about memos

```
ship              part_of           hull

a_kind_of                                                 a_kind_of

enterprise                part_of           enterprise_hull
```

Figure 12-2: an example of part-kind inheritance

```
                          physical_object

                                  a_kind_of

                          vehicle

                                  a_kind_of
```

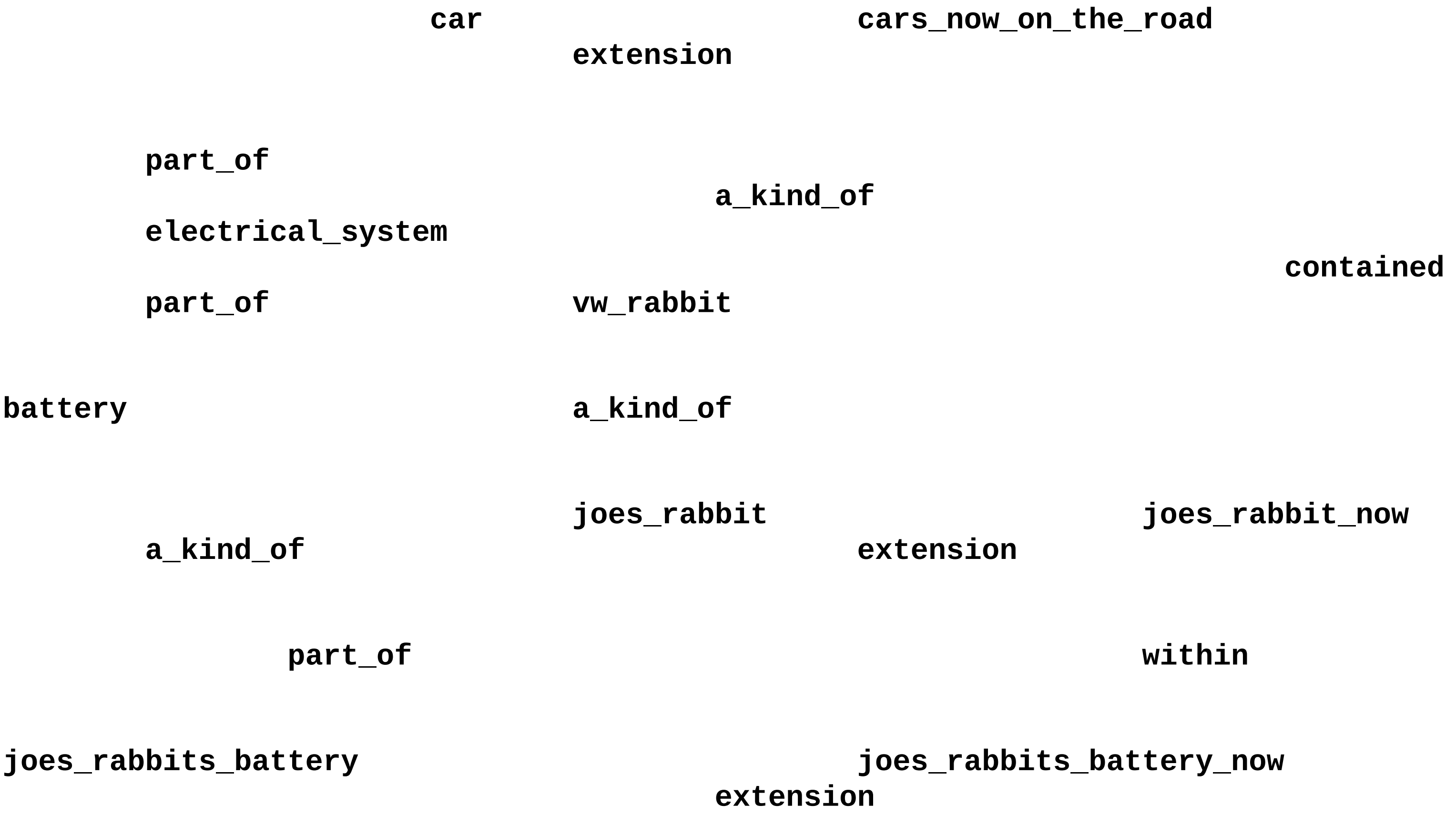


Figure 12-3: the semantic network for the frames in the cars example

```
frame_name: sending8347
a_kind_of: sending
actor: we
object: memo72185
time: yesterday
destination: headquarters
method: express mail
```

```
frame_name: memo72185
a_kind_of: memo
author: Tom
addressees: [Ann]
origin: drawingup20991
```

```
frame_name: drawingup20991
a_kind_of: drawingup
actor: Tom
object: memo72185
beneficiary: Ann
time: 6/12
```

.sp 7

Figure 12-4: frame representation of the meaning (semantics) of the sentence "Yesterday we sent headquarters by express mail the budget memo that Tom drew up for Ann on 6/12"

```
                                                anyone
                                                on anything
                                                anytime

                        anyone          anyone          Tom's group
                        on languages    on anything     on anything
                        anytime         today                   anytime

anyone          anyone          Tom's group     Tom's group     Tom
on Prolog               on languages    on languages    on anything     on anything
anytime         today                   anytime         today
anytime

anyone          Tom's group     Tom's group     Tom                     Tom
on Prolog               on Prolog               on languages    on languages    on
anything
today                   anytime         today                   anytime         today

                        Tom's group     Tom                     Tom
                        on Prolog               on Prolog               on languages
                        today                   anytime         today

                                                Tom
                                                on Prolog
                                                today
```

Note: "languages" means "programming languages", "Prolog" means "Prolog interpreter"; all arrows represent a_kind_of

Figure 12-5: a three-dimensional inheritance hierarchy of user model frames

---

Time|Monday|Tuesday|Wednesday|Thursday|Friday

9|occupied|occupied|occupied||occupied 10|occupied|occupied|occupied|occupied|occupied 11|class|occupied|occupied|class|occupied 12|occupied|occupied|occupied|occupied|occupied 1|class|class|occupied|class|occupied 2|occupied||occupied||occupied 3|occupied|occupied|occupied|occupied|occupied

4	**occupied**	**occupied**	**occupied**		**occupied**

Figure 13-1: an example scheduling problem, with an example solution symbolized by "class"

---

value	*Satisfies label*?	*Satisfies* **large?**	*Satisfies* **regular?**	*Satisfies* **irregular?**
grass	**yes**	**yes**	**no**	**yes**
water	**yes**	**yes**	**no**	**yes**
pavement	**yes**	**yes**	**yes**	**no**
house	**yes**	**yes**	**yes**	**no**
vehicle	**yes**	**no**	**yes**	**no**

Figure 13-3: summary of the single-argument predicates in the photo interpretation example

---

A1	*A2*	*Satisfies*	*Satisfies*
value	*value*	*borders(A1,A2)*?	**inside(A1,A2)?**
grass	**grass**	**no**	**no**
grass	**water**	**yes**	**yes**
grass	**pavement**	**yes**	**yes**
grass	**house**	**yes**	**no**
grass	**vehicle**	**no**	**no**
water	**grass**	**yes**	**yes**
water	**water**	**no**	**no**
water	**pavement**	**yes**	**yes**
water	**house**	**no**	**no**
water	**vehicle**	**no**	**no**
pavement	**grass**	**yes**	**no**
pavement	**water**	**yes**	**no**
pavement	**pavement**	**no**	**no**
pavement	**house**	**yes**	**no**
pavement	**vehicle**	**yes**	**no**
house	**grass**	**yes**	**yes**
house	**water**	**no**	**no**
house	**pavement**	**yes**	**yes**
house	**house**	**no**	**no**
house	**vehicle**	**no**	**no**
vehicle	**grass**	**no**	**no**
vehicle	**water**	**no**	**no**
vehicle	**pavement**	**yes**	**yes**
vehicle	**house**	**no**	**no**
vehicle	**vehicle**	**no**	**no**

Figure 13-4: summary of the two-argument predicates in the photo interpretation example

---

Given the query:

```
?- a(X), b(X,Y), c(Z), d(Y,Z), e(X).
```

These are its dependencies:

Predicate	*Variables*	*Dependencies to*	*Dependencies that*
expression	*bound*	*previous predicate*	*bind a common variable*
		expressions	
a(X)	X	none	none
b(X,Y)	Y	a(X)	a(X)
c(Z)	Z	none	none
d(Y,Z)	none	b(X,Y),c(Z)	b(X,Y),c(Z)
e(X)	none	a(X),b(X,Y)	a(X)

Figure 13-5: summary of dependencies in an example query

Given the query:

```
?- a(X), b(X,Y), c(Z), d(X,Z), e(X).
```

Given the database:

```
a(1).
a(2).
a(3).
b(A,B) :- B is 3*A.
c(4).
c(1).
d(A,B) :- A>B.
e(3).
```

This is what happens in dependency-based backtracking:

Step	*a(X)*	*b(X,Y)*	*c(Z)*	*d(X,Z)*	*e(X)*
start	active	active	active	active	active
1	X bound to 1	active	active	active	active
2	inactive	Y bound to 3	active	active	active
3	inactive	inactive	Z bound to 4	active	active
4	inactive	inactive	inactive	fails	active
5	active	inactive	Z bound to 1	active	active
6	active	inactive	inactive	fails	active
7	active	inactive	fails	active	active
8	X bound to 2	active	active	active	active
9	inactive	Y bound to 6	active	active	active
10	inactive	inactive	Z bound to 4	active	active
11	inactive	inactive	inactive	fails	active
12	active	inactive	Z bound to 1	active	active
13	active	inactive	inactive	succeeds	active
14	active	inactive	inactive	inactive	fails
15	X bound to 3	active	inactive	active	active
16	inactive	Y bound to 9	inactive	active	active
17	inactive	inactive	inactive	succeeds	active
18	inactive	inactive	inactive	inactive	succeeds

Figure 13-6: a dependency-based backtracking example

---

n	*p*	*Probability*
5	.25	.76
10	.25	.94
20	.25	1.0E+00
50	.25	1.0E+00
5	.1	.4
10	.1	.65
20	.1	.88
50	.1	.99
5	.01	.05
10	.01	.10
20	.01	.18
50	.01	.39

100|.01|.63

Figure 13-7: the probability that some item in a possibility list will be impossible, for an n-item list with probability p that a possibility will be impossible

---

```
                                 relax

                setup            relax2


repeatactive    possible_value           update_activity check_unique


                constraint_violation                     some_var


                substitute


                                         member  one_possibility


        choices          constraint              satisfiable
        [user-written]   [user-written]          [user-written]

                                                                          some_bindings
```

Figure 13-8: the predicate hierarchy for the pure-relaxation program

---

a|b|a; not(b)|Is the rule|Justification
|||a :- b. consistent
|||(uncontradicted)?

true|true|true|yes|b proves a

true|false|true|yes|a could be proved ||||by another rule, and ||||that wouldn't violate ||||the truth of this one

false|true|false|no|b proves a by the ||||rule, a contradiction

false|false|true|yes|the rule doesn't ||||apply, and there may ||||be no other rule to ||||prove a

Figure 14-1: demonstration that an equivalent logical (declarative) meaning of the Prolog rule a :- b. is a; not(b).

---

a; not(b); not(c).   c; not(d).   b.   d.

not(a).
[from step 1]

not(b); not(c).
[from step 2]

not(c).
[from step 3]

not(d).
[from step 4]

"null clause"
[from step 5]

Figure 14-2: a resolution example

In rule form|In clause form

BACKWARDS CHAINING:| QUERY: ?- a.|ASSUMPTION: not(a). a :- b, c, d.|a; not(b); not(c); not(d). **RESULTS IN:|RESOLVES TO: ?-b, c, d.|not(b); not(c); not(d). FORWARDS CHAINING:| a :- b, c.|a; not(b); not(c). c.|c. RESULTS IN:|RESOLVES TO: a :- b.|a; not(b). RULE COLLAPSING:| a :- b, c, d.|a; not(b); not(c); not(d). c :- e, f.|c; not(e); not(f). RESULTS IN:|RESOLVES TO: a :- b, e, f, d.|*****a; not(b); not(e); not(f); not(d).***

Figure 14-3: three special cases of resolution

---

1. methods requiring a gold standard

(a) traces
(b) confusion matrices
(c) comparison of a near miss and example

(i) no numbers involved
(ii) composite results
(iii) numbers involved

(d) comparison of two examples

(i) no numbers involved
(ii) composite results
(iii) numbers involved

2. methods not requiring a gold standard

(a) schema for rules and facts
(b) user-initiated debugging questions
(c) cooperativeness evaluation
(d) a priori suitability for artificial intelligence techniques

Figure 15-1: tools for testing and debugging

```
      guessed|battery|electrical|fuel|engine
       cause|dead|short|blockage|problem
actual
cause
battery dead
```

|42|1|4|0 electrical short|15|29|8|3 fuel blockage|0|2|25|1 engine problem|1|0|8|12

Figure 15-2: a confusion matrix, for an expert system diagnosing situations in which a car won't start

---

Given the rule a :- b, c.
Assume a, b, c, d, e, f, and g are predicate expressions.
Assume b, c, d, and e are composed of one or more predicate expressions "and"ed and "or"d together.

```
Nature of|Predicate expressions|Suggested new rule
the new case|that are true
|in the new case (those
|not mentioned are false)
```

example|a and b and c|no change needed example|a and b|a :- b. example|a and b and c and d|no change needed example|a and b and d|a :- b, e. ||where e is implied by both c and d ||(if no such e, try a:- b.) example|f and b and c|g :- b, c. ||where g is implied by both a and f ||(if no such g, create a new rule f :- b, c.) near miss|a and b and c|remove rule entirely near miss|a and b|no change needed near miss|a and b and c and d|a :- b, c, not(d). near miss|a and b and d|no

change needed near miss|f and b and c|no change needed

Figure 15-3: suggested rule modifications given a new case

A|B|A AND B|A OR B|NOT A|

true|true|true|true|false true|false|false|true|false false|true|false|true|true false|false|false|false|true Figure A-1: summary of AND, OR, and NOT

**Task: add 1000 to each salary in a set of 732
employee salaries, starting with employee #1**

**Task: add 100 to the Task: add 1000 to each salary in a set of 731
salary of employee #1 employee salaries, starting with employee #2**

**Task: add 1000 to the Task: add 1000 to each salary in a set of 730
salary of employee #2 employee salaries, starting with employee #3**

**Task: add 1000 to each salary in a set of 732
employee salaries, starting with employee #1**

**Task: add 1000 to each salary in Task: add 1000 to each salary in
a set of 366 employee salaries, a set of 366 employee salaries,
starting with employee #1 starting with employee #367**

**Task: add 1000 Task: add 1000 Task: add 1000 Task: add 1000
to each salary to each salary to each salary to each salary
in a set of 183 in a set of 183 in a set of 183 in a set of 183
starting with starting with starting with starting with
employee #1 employee #184 employee #367 employee #550**

Figure B-1: two different ways of doing an example recursion

**Task: take the sum of 732 employee
salaries, starting with employee #1**

(the right box below calculates the result)

**Task: take the sum of 731 employee Task: add the result in the left box
salaries, starting with employee #2 to the salary of employee #1**

(the right box below calculates the result)

**Task: take the sum of 730 employee Task: add the result in the left box
salaries, starting with employee #3 to the salary of employee #2**

Figure B-2: an example of recursion of a function

array: list: stack using an array (insert and delete only at right end): queue using an array (insert at left end, delete at right end): tree: .sp 8

lattice: .sp 8

graph: .sp 8

Figure C-1: pictures of examples of some famous data structures

```
Built-in|Arguments|Can it succeed on|Description
predicate||backtracking?
```

consult|a filename|no|loads a file listing|a predicate name|no|prints database items |||with that predicate name asserta|a fact|no|adds a fact to the front |||of the database assertz|a fact|no|adds a fact to the rear |||of the database retract|a fact|no|removes a fact abolish|a predicate name|no|removes all facts with |and its number||the same predicate name |of arguments| > [infix]|two numbers|no|greater-than < [infix]|two numbers|no|less-than = [infix]|two numbers|yes|equals is [infix]|variable and|yes|arithmetic assignment |a numeric expression| number|a variable|no|succeeds if argument |||is bound to a number write|a variable or|no|prints on the terminal |character string read|a variable or|no|reads a word typed |character string||by the user nl|none|no|issues carriage return to |||the terminal not|a predicate|no|succeeds if querying its |expression||argument fails fail|none|no|always fails !|none|no|[see text] var|a variable|no|succeeds if argument |||is not bound to anything call|a predicate|yes|queries the predicate |expression clause|a rule left side|yes|binds text of a rule |and rule right side||to any variables =.. [infix]|a query|yes|converts a query to a |and a list||list or vice versa

Figure D-1: summary of built-in Prolog predicates assumed in this book

```
Standard Prolog|Micro-Prolog
feature|feature
```

consult(<filename>)|LOAD <filename> listing(<predicatename>)|LIST <predicatename> asserta(F)|(ADDCL F 0) assertz(F)|(ADDCL F <number-of-F-facts-and-rules>) retract(F)|(DELCL F) abolish(F,2)|KILL F ?- |&. :-|[see discussion] ,|[see discussion] p(X); q(X)|(OR ((p X)) ((q X))) not(p(X))|(NOT p X) X<Y|(LESS X Y) X>Y|(LESS Y X) X=3|(EQ X 3) X is 3|(EQ X 3) X is Y+Z|(SUM Y Z X) X is Y-Z|(SUM X Z Y) X is Y*Z|(TIMES Y Z X) X is Y/Z|(TIMES X Z Y) [a,b,X]|(a b X) [X|Y]|(X | Y) 'Character string'|(Character string) write(X)|(P X) write(X),nl|(PP X) read(X)|(R X) clause(p(X),Y)|((p X) | Y) P=..L|[unnecessary--queries are lists] call(p(X))|(? ((p X)) fail|(FAIL) !|! number(X)|(NUM X) var(X)|(VAR X) /* comment */|/* comment setof(X,p(X),X2)|(ISALL X2 X (p X))

Figure E-1: approximate equivalents between the Prolog assumed in this book and Micro-Prolog

a

Figure G-1: answer to problem 9-11.(a)

Go to paper index

ADVICE TO THE INSTRUCTOR

If artificial intelligence is a hard subject to learn, it's an even harder subject to teach. Intellectually, it's one of the most difficult areas of computer science, with many complicated ideas to understand. It requires a good deal of background. It's very rigorous in some subareas, and sloppy in others. It's not like any other subject, so students have few analogies on which to draw. Instead, they see and hear distorted media reports about it that encourage unreasonable expectations. So instructors of artificial intelligence courses have their hands full.

But on the other hand, if you can successfully teach an artificial intelligence course in the style and at the level of this book, you've really accomplished something. The students will almost certainly learn a lot. And much of what they'll learn is quite practical, and much of it will tie in with important concepts they've learned in other courses. So don't be easily discouraged.

Evaluating Student Progress

One major difficulty in teaching artificial intelligence is in planning the test questions and homework exercises. Many instructors have told me how conventional tests just don't seem to work for artificial intelligence courses. The interesting questions that really test if students know anything require more than an hour to do, and multi-hour tests are hard to arrange and unfair to students who just can't show creativity under the pressure of a test. Substantial homework assignments seem the only alternative, and in fact, most of the questions in this book are intended as homework questions. So don't be afraid to give lots of homework if you have the capability to grade it.

Homework has a potentially big disadvantage: the possibility of collaboration or copying of answers between students. You'll just have to judge how worrisome this is for your institution. At MIT in 1973, in my first course in this subject, Patrick Winston deliberately encouraged students to work together on the all-day take-home exams, claiming that grades in the course weren't that important. At Stanford in 1981, when I assisted with the first course, we didn't issue any policy on the homework, and trusted the student's adherence to the Honor Code, which a few students flagrantly violated. At the Naval Postgraduate School, where I grade all papers myself, I make a special effort to compare papers on the first homework assignment, and warn students whose papers seem to be significantly more similar than could be due to pure chance. But I can't say that my experience has been typical.

Still, you have to give some tests. Currently, in my 11-week quarter course (introductory artificial intelligence) I give a 50-minute midterm (with about four questions) and a 110-minute final exam (without about seven questions). The easier exercises in the book, especially those involving following of algorithms, are generally my test questions. I also give three or four homework assignments, evenly spaced through the quarter. Currently, I get an average of about 70% correct on the homework and tests. I recommend open-book tests because there are many terms used in the field, and the material is so hard anyway for students.

About Good Exercises and Test Questions

I suppose some of you are thinking that it surely must be possible to construct some easier test questions--maybe multiple choice?--than those in this book. I'll argue that it's very difficult to do this and remain fair to the content of AI, because the subject concerns thinking, and you need students to think on a test to verify that they see how computers could do it. Let's examine some question types that are not in my book.

Q1. Define resolution.

That just requires memory regurgitation; there's no assurance that correct answer means the student understands what resolution requires or what it's good for. This sort of bad question is the too-frequent resort of lazy professors.

Q2. Which of these is a control structure: Bayes' Rule, caching, unification, or forward chaining?

Again, mostly memory regurgitation is needed. And it's too easy a question; if students are really so confused that they haven't realized that forward chaining is a control structure, then either standards were too low, the course was really awful, or the students shouldn't be in college. Why not instead kill several birds with one stone, and ask them a more complicated question about forward chaining that they won't make any progress on unless they know it's a control structure, like "Can there be a breadth-first forward chaining?"

Q3. Implement in a computer program the idea expressed on page 76, paragraph 3, line 5.

Learning computer science means doing some programming, but not necessarily a lot, and not necessarily in every course. Though my book uses Prolog, there aren't many programming exercises for the reason that such exercises can require unexpected amounts of student time and effort. I think there are better and faster ways for students to learn the material, ways that do not involve programming, just in the same way there are better ways to learn physics than trying to duplicate every famous experiment in the lab. And also, programming exercises easy for the professor to write like the one above are rarely the most valuable programming experiences for students.

Q4. Discuss the difference between breadth-first search and best-first search.

Broad essay questions can be trouble to grade because they give the student a lot of room to pontificate, and it can be hard to compare student performances when different students cover different points. And it's not clear what level of analysis is an adequate answer to the question, so you'll frequently have students in your office grumbling if you give questions like this.

Q5. Prove that if A implies B or C, and B implies A, then C is true.

Leave proofs for math classes; mathematics instructors know better how to teach proof rigor, and have more reason to teach it. And there's more than enough material to teach in artificial intelligence that other courses don't teach.

Q6. Suzy Seashell is preparing a Honorary Dinner and needs to select a wine. Her choices are Chateau Aulde-Tenni-Chue, Rotgutter Dranow, and Cheap Yellow. Her expert system has rules as follows: ...

OK, I confess I have a few silly questions in my book, to get back at certain peeves. But if there are too many such questions, students lose track of what the course is for, and eventually lose their motivation. A diet of desserts is not good for the health.

Demonstrations

If at all possible, try to use an actual computer during class to show off the programs of the book and other similar programs. It will make some things much clearer for the students. This is particularly valuable at the beginning of the course, but it works well for some of the later programs in the book too; I'd recommend doing a demonstration once every eight classroom hours. To make this easier, Prentice-Hall is selling tapes and diskettes containing the major programs in the book; see your representative. Devices for projecting computer-terminal screens on a lecture-hall screen are an invaluable for computer science courses, if you haven't tried them before.

A Sample Syllabus

Following is my syllabus for when I gave a quarter course "Artificial Intelligence" based on this book, as the only text, in Summer Quarter 1987. Our quarter was 11 weeks long, and the course met four times a week for 50-minute lecture/recitations (with zero lab hours allocated). Thirty students per quarter has been my average enrollment, and I grade all homework and tests myself.

I have also used other parts of the book in an advanced course, "Expert Systems". I usually use Chapters 7 and 10, plus an expert-systems book. Careful coverage of those two chapters could almost fill a quarter by itself, since many issues in Prolog programming as well as expert systems are raised by those chapters.

###

CS3310, Artificial Intelligence

Summer 1987

Profs. Rowe (Spanagel 528B) and McGhee (Spanagel 512)

Official office hours for Prof. Rowe: Mondays, Wednesdays, and Thursdays 9-10 and 11-12. No official hours for Prof. McGhee; see him anytime.

Grading: 75 points on each of four Homework assignments; 100 points Midterm Exam (open-book); 175 points Final Exam (open-book).

Both sections of this course will cover the same material, and they will have the same homework and tests.

The listed prerequisite for this course is a course in logic, such as Mathematics 2025 or 0125 at this school, but this isn't too important. But it is important to have taken at least one college-level programming course, hopefully two. A data structures course helps considerably, as do software engineering courses.

This course will require at least one hour of independent study outside of class every day (six days a week). The homework and tests will be difficult, with expected class averages around 70%. Questions will emphasize thinking more than memorization, and will often involve applying what you know to new applications. This is definitely a graduate-level course.

The textbook is a set of class notes. For additional reading (not required) see Winston, Artificial Intelligence, second edition (Addison-Wesley, 1984), or the other books listed in the bibliography in the notes (Appendix F).

For the last six terms of this course, 36 A grades were given, 38 A-, 65 B+, 53 B, 22 B-, 3 C+, and 4 C grades.

Schedule (do not hand in the practice problems):

By 7/9: read chapters 1 and 2 of the notes; do for practice problems 2-2 and 2-6

By 7/15: read chapter 3 (look over Appendix A if you need to review logic); do for practice problem 3-9

By 7/22: read chapter 4; do for practice problem 4-11

Homework #1 due Thursday 7/23

By 7/29: read chapter 5 (look over Appendices B and C to review recursion and data structures); do for practice problems 5-2 and 5-14

By 8/5: read chapter 6, sections 6.1-6.6 and 6.8 only; do for practice problem 6-5

Homework #2 due Monday 8/10

By 8/11: read chapter 7, sections 7.1-7.9 only; do for practice problem 7-2

Midterm exam Thursday 8/13 (covering chapters 1-7)

By 8/19: read chapter 8, sections 8.1-8.10 only; do for practice problem 8-3

By 8/25: read chapter 9 except for sections 9.14-9.16; do for practice problem 9-11

By 8/31: read chapter 10, sections 10.1-10.4, 10.6, and 10.7 only; do for practice problem 10-3

Homework #3 due Tuesday 9/1

By 9/7: read chapter 11, sections 11.1-11.7 only; do for practice problem 11-6

By 9/10: read chapter 12; do for practice problem 12-5

By 9/15: read chapter 15

Homework #4 due Wednesday 9/16

Final exam Monday, 9/21, 0800, Sp-421

Alternative Syllabi

Despite the statements of the Preface (which are designed to encourage people not to skip chapters), flexibility is possible in which chapters are taught. But please try not to skip Chapter 15, which is important.

More and more Prolog courses are being offered at schools. If a Prolog course is required or common for your students, there is still more than enough material in this book for an AI course. Certainly, you can go more quickly through the first five chapters of this book. However, definitely do not skip those chapters entirely, because they contain many important AI ideas not usually covered in Prolog texts.

Some of you may prefer a more formal approach based on logic, perhaps because you teach in a mathematics department or come from a more theoretical background. Then you can take Chapter 14 early, perhaps right after Chapter 3; Chapter 14 has been designed so it makes minimal references to the rest of the book. You may want to skip the more pragmatic parts of the book, like Chapters 7 and 10. However, if you want to take a formal approach, consider the following argument. Students can't be motivated to learn the syntax and manipulations of logic unless they can see what it is used for in practical applications. My approach of emphasizing the semantics of predicate expression from the very beginning, bringing up formal details slowly and only as absolutely necessary, tends I believe to motivate students more.

Some of you would prefer to spend some time on the specifics of some subarea of AI; natural language seems to be the most commonly mentioned subarea. My book does try to stick with subarea-independent concepts, but actually, there is a lot of natural language material in my book, mixed in with others things as per my integrated approach; note especially sections 2.8, 4.4, 5.9, 6.12, and 12.13. You may want to supplement this with additional material. But before you do, consider the following. Linguistics is an important field of scientific study, but claims of important practical benefits for it have never panned out. There seems little need to have computers communicate with us in English any more, what with major recent advances in graphics, positional input devices, and menu-driven systems. Words take a long time to type, and many people can't type including many who are illiterate, so it's unfair to emphasize words in using computers; an emphasis on words may just be a relic of the overly verbal education most computer researchers receive. There is still some need to process written text by computer, but the increasing computerization of every aspect of human life is making it progressively unimportant. Emphasizing language in an AI course also means teaching science instead of engineering, and I don't think this is what students need in a university: there are too many science courses already.

Some reviewers of the manuscript complained about various omissions and placements of Prolog features in the book. Please understand I'm not trying to teach Prolog; many books available can do that, and it really requires a full quarter of instruction. Instead, I'm trying to use Prolog to illustrate AI concepts, and I'm very concerned about introducing too much too fast. Some specific issues:

--The anonymous variable ("_") is omitted because I believe it encourages poor programming. Intelligent naming of variables is an important part of programming (see section 2.4), an students shouldn't be allowed to avoid or "cop out" on it.

--It's a lot easier to explain the "not" construct by pretending it's a predicate that takes a predicate expression as input, though this isn't the way it's implemented in many Prologs. Explaining it any other way would needlessly complicate the early chapters of the book.

--I ignore ways of entering facts directly into the Prolog interpreter's database (like "data-entry mode"), for a philosophical reason: I think students need to have emphasized the distinction between knowledge (facts and rules) and control structures. If you muddy this distinction, students can get very confused. It's also a distinction important historically in AI.

--The cut predicate is terribly confusing for students, and it's much abused in programming (I was amazed to see a recent Prolog text demonstrate expert systems with a cut symbol at the end of every rule, horrendous programming style), so I postpone it to a place where its complexities are an appropriate match to the material, in Chapter 10. I fervently hope you'll resist the temptation to introduce it earlier to your students, because there's plenty of more important things in AI to confuse them. However, some of the software accompanying this book contains cuts for efficiency, when the versions in the book don't have cuts.

--I do not use complex structures very much in this book (that is, embedded predicate expressions). I hope you won't either, because it is important for an AI course to maintain some compatibility with Lisp. Embedded lists can do everything that complex structures can, and in a way more analogous to Lisp, so I've used them instead in this book (for instance, in the search path lists of Chapter 10). Nearly all students who continue on to study AI will learn Lisp.

--I haven't been afraid to give some things that don't work in Turbo Prolog. Turbo has generated a lot of initial interest based on its price, but it's unclear how popular it will be in the long run. Its compiler-based approach means quite different processing from interpreted Prologs, and it has a number of important incompatibilities with Clocksin and Mellish Prolog, despite claims of its source, most notably in the "funarg" problem and other issues in the persistence of variable bindings. Several important and interesting programs like forall and doall in Chapter 7 won't work in Turbo, but I feel they're important enough (and relate directly to counterparts in Lisp) that students should see them. Cheaper and simpler dialects of Prolog are desirable, but they shouldn't leave out key features.

.PA

ADDITIONAL PROBLEMS

Note: answers to these questions are at the back of this Manual.

I-1. (for Chapter 3) Consider the Prolog database:

a(X) :- b(X,Y), b(Y,Z). b(1,3). b(2,2). b(4,2).

(a) What will a Prolog interpreter answer to the query:

?- a(2).

(b) List all the possible answers that a Prolog interpreter could find to the following query to this database (not including "no"), if you kept typing semicolons:

?- a(Q), not(b(Q,Q)).

(c) How many times total does the Prolog interpreter backtrack from b(Y,Z) to b(X,Y) in the rule, when you keep typing semicolons, before it answers "no" to the query of part (b)?

I-2 (for Chapter 3). Consider the query:

?- link(A,B), link(B,C).

Suppose there are four "link" facts in the Prolog database. Assume no duplicate facts. Assume the two arguments are never identical within a fact.

(a) What is the maximum number of answers a Prolog interpreter will find for the this query?

(b) What is the maximum number of times a Prolog interpreter will backtrack from the second predicate expression to the first predicate expression before answering?

I-3. (for Chapter 3) Why do these two Prolog queries always give the same set of answers?

?- (a;not(c)), b, c. ?- a, b, c.

I-4 (for Chapter 3). (a) Sometimes unbound variables inside "not"s work OK. Consider this query:

?- bosses(R,S), bosses(T,R), not(bosses(S,U)).

with the database

bosses(poindexter,north). bosses(reagan,poindexter). bosses(north,hall).

What is the first answer the Prolog interpreter will find to the query? (Hint: draw a semantic network.)

(b) For the query and database of part (b), how many times will Prolog backtrack from the second predicate expression to the first in getting the first answer?

I-5. (for Chapter 3) Suppose a Prolog query has 26 answers total (obtained by typing semicolons). Suppose this query is the "and" of some predicate expressions, with no "or"s (semicolons) or "not"s.

(a) Suppose you give the Prolog interpreter a new query just like the old except with another predicate expression "and"ed on the right end. Suppose this new expression doesn't contain any "new" variables, variables not occurring in the old query. Suppose this augmented query has 26 answers too. Must they be the same answers? Why?

(b) For the same situation as (a), suppose we measure the CPU (processing) time to get all possible answers, measuring up until the interpreter types "no". Will the new query take more time than the original query, less time, or sometimes more and sometimes less depending on the query? Why?

(c) Repeat question (b) but assume the new query has 7 answers total (with the original query still having 26). Why?

(d) Repeat question (c) but assume the new predicate expression was "and"ed on the left end of the query, not the right. (Still assume 7 answers total.)

I-6. (for Chapter 4) (a) Which of the following (pick only one) is a Prolog rule for downward inheritance of "owns" with respect to the part-of relationship? ("owns(P,T)" means P owns thing T.)

part_of(P,T) :- owns(P,Q), part_of(Q,T). part_of(P,T) :- owns(U,T), part_of(U,P). part_of(P,T) :- part_of(T,Q), owns(P,T). owns(P,T) :- part_of(Q,P), owns(Q,T). owns(P,T) :- part_of(U,T), owns(U,P). owns(P,T) :- owns(P,Q), owns(Q,T).

(b) Suppose the U.S. Navy owns the Enterprise. Hence the U.S. Government owns the Enterprise; does that follow from the inheritance rule you selected above? If so, what do the variables match? If not, why not?

I-7. (for Chapter 4) Suppose predicate "necessary(X,Y)" means that a specific single action X must occur before specific single action Y can occur.

(a) Is predicate "necessary" transitive?

(b) Consider the predicate "before9(X)" which is true if specific single action X occurred before 9AM on February 10, 1986. Does "before" inherit with respect to "necessary" in any way? If so, describe how. (Hint: "necessary" can be used backwards.)

I-8. (for Chapter 5) This "delete" predicate takes three arguments: an item, a list, and a list. It says that the result of deleting all occurrences of the item from the first list gives the second list.

delete(X,[],[]). delete(X,[X|L],L2) :- delete(X,L,L2). delete(X,[Y|L],[Y|L2]) :- delete(X,L,L2), not(X=Y).

Consider the query:

?- delete(X,[a,b,c],[a,c]).

(a) What is the first recursive call of "delete"? Identify the arguments of the call.

(b) What answer does the original query (the one printed above) give? (Hint: think what the definition means.)

I-9. (for Chapter 5). Define a predicate "pair(L,PL)" which takes a list of even length as first argument, pairs the adjacent items in the list, and binds that result to its second argument. So for instance:

?- pair([a,b,c,d,e,f],P). P=[[a,b],[c,d],[e,f]]

Show your program working.

I-10. (for Chapter 10) Define a predicate "intersection(S1,S2,I)" which says that set I is the intersection of sets S1 and S2, where all three sets are expressed as lists. (That is, items in I are all the items that occur within both S1 and S2.) You can use any of the definitions of chapter 5 to make the definition simpler, but don't use the "var" predicate. Show your program works correctly for two cases: (1) for S1 and S2 bound, I unbound; (2) for S1 and S2 unbound, I bound. (Remember, things Prolog prints out that start with an underscore symbol ("_") are Prolog-invented variable names.)

I-11. (for Chapter 5) In 20 words or less, give the declarative meaning of program "f".

```
f(L,M) :- g(L,[],L2). g([],L,L). g([X|L],L2,L3) :- g(L,[X|L2],L3).
```

I-12. (for Chapter 6) Consider the rule:

```
bottom(S) :- bosses(R,S), not(bosses(S,U)).
```

with the facts:

```
bosses(poindexter,north). bosses(reagan,poindexter). bosses(north,hall).
```

Do pure forward chaining. Give every new rule the algorithm creates, and give every new fact it proves, until it runs out of things to do. Use focus-of-attention placement of new facts.

I-13. (for Chapter 6) Consider this database of rules and facts:

```
a(X) :- b(X), c(X). c(Longvariable) :- f(Longvariable). c(2) :- not(e(Z)). b(Y) :- c(Y), not(d(Y)). e(5). f(3).
```

(a) What rule rewritings are done and what facts are proved, in order, by pure forward chaining with the focus-of-attention approach? Assume you want all possible facts. Remember, save "not"s for last.

(b) What facts are proved, in order, by rule-cycle hybrid chaining with the focus-of-attention approach? Also give the cycle number on which the fact was proved. Assume you want all possible facts. Remember, save "not"s for last.

I-14 (for Chapter 6). Consider the following rules for diagnosis of childhood illnesses:

R1: If there's a rash and but not a high fever, suspect chicken pox. R2: If there's a rash, a cold symptom, and a mild fever, suspect German measles. R3: If there's a rash, a cold symptom, and a high fever, suspect measles. R4: If there's a mild fever and sore throat, suspect mumps. R5: If there's a medium fever and sore throat, suspect scarlet fever. R6: If there's a cold symptom and any fever, suspect a cold. R7: A high fever is over 102. R8: A medium fever is over 100.5 to 102. R9: A mild fever is 99 to 100.5. R10: Sneezing is a cold symptom. R11: Headache is a cold symptom. R12: Sore throat is a cold symptom.

(a) Write the above as twelve Prolog rules. Use variables where appropriate. (Hint: they are appropriate somewhere.)

(b) Suppose we do backwards chaining on these rules with the goals taken in order to be measles, scarlet fever, chicken pox, mumps, German measles, cold. Suppose the facts in order are a sore throat, a temperature of 101, a desire to go to the circus, and a rash. Suppose we continue until everything provable is proved. What conclusions are reached (including intermediate ones), in order?

(c) Suppose we do pure forward chaining with the same specifications as part (b), using focus-of-attention placement of new facts. Suppose we continue until everything provable is proved. What conclusions (including intermediate) are reached, in order?

(d) Suppose we do rule-cycle hybrid chaining with the same specifications as part (b). Suppose we continue until everything provable is proved. What conclusions (including intermediate) are reached, in order?

(e) What is the advantage of taking the disease conclusions in the order given in part (b)? That is, why that order and not some other?

(f) Would a decision lattice be a good way to compile a set of medical-diagnosis rules like these, a set say of about 1000 rules for common problems? Why?

(g) Would "and" parallelism implemented with multiple processors be a good idea with a set of medical-diagnosis rules like these, assuming the same 1000 common-problem rules as part (f)? Why? Think of how the rule-based would actually be used.

I-15. (for Chapter 6) Suppose you have two processors that access the same Prolog database, both of which can run Prolog independently. Describe all the useful possible concurrencies (including full details of what should be run in parallel with what) in the following code:

```
p(X) :- q(X), r(X,Y), s(X), t(Y). p(X) :- u(X).
```

when used with the following query:

```
?- p(Q).
```

In particular, explain carefully how maximal and-parallelism could be done.

I-16. (for Chapter 6) Suppose a grammar is written in the manner of section 5.9, as substitutions of lists of words for other lists of words. Substitutions are not "coupled", so two substitutions might make sense individually, but not when used together, at two different places in the same sentence. How could a Prolog implementation apply "semantic" criteria to rule out such nonmeaningful combinations? Give implementation details.

I-17. (for Chapter 6) (a) Which control structure is a decision lattice most similar to, backward chaining or forward chaining? Explain.

(b) Which control structure is an and-or-not lattice most similar to, backward chaining or forward chaining? Explain.

I-18. (for Chapter 7) Study Exercise 7-5 and its answer.

(a) Now suppose the original code we have for testing whether a(X) is true is in the form of M rules, each requiring computation cost of 2H/M. Suppose the rules are tried sequentially until one succeeds, just like cached facts. Suppose each has a mutually exclusive probability Q of success. Redo part (a) of the original problem to get a new formula for when using a cache will be preferable.

(b) Now suppose the rules are not equally costly, but are arranged from least costly to most costly. Suppose further the costs increase linearly so that the Ith rule costs about 4HI/M*M. Redo part (a) of problem 7-5.

I-19. (for Chapter 7) The appliance program (via the "ask" code) caches the answers to all questions it asks the user. Would any additional caching (besides that of final answers, which could prevent repeated printing of the same diagnosis) significantly improve the program?

I-20. (for Chapter 7) Explain how forward chaining must be modified to handle rules with arithmetic computations, where the computations are used only to compute numbers for the left sides of the rules.

I-21. (for Chapter 7) (for interpreted Prologs only) The language Lisp has a "prog" construct, whereby a list of statements are executed in order. Define the equivalent for Prolog. That is, define a predicate "prog" of one argument, a list of predicate expressions, that calls every one of the expressions in order, ignoring whether any expression succeeds or fails. It also should not permit backtracking from expression to expression if something after the "prog" fails. Hint: you can't backtrack into a "not".

I-22. (for Chapter 7) Construct a rule-based system that models an elevator that you have access to. Conduct experiments to find out what the elevator does in particular situations. Then test your program in those same situations to confirm the same behavior. Use as program input information about which buttons were pushed when, where the other elevator is, whether someone is standing in the door, etc. Let the program output be a statement about elevator behavior, either (1) it goes to a particular floor, or (2) its doors open or close, or (3) it stops, or (4) its alarm bell rings, etc.

I-23. (for Chapter 7) Find a misleading advertisement for an expert-system software product. Point out what is misleading. Beyond the hype, what do you each is really selling, in the terminology of this chapter?

I-24. (for Chapter 8) Suppose you're writing an expert system to detect Russian submarines.

--Suppose an expert tells you that if there are Russian manuevers in the area, the probability is 0.5 that any submarine waveform you detect will be a Russian submarine.

--Suppose the expert also says that if you see a type Z37 waveform, it's a Russian submarine with probability 0.8.

(a) Now suppose an expert says that if (i) there are Russian manuevers in the area, and (ii) you think with probability 0.5 that you see a Z37 waveform (that is, you're only half sure that the waveform you see go by on the screen was a Z37), then the probability it's a Russian submarine is 0.7. What evidence combination method is the expert using: independence-assumption, conservative, liberal, or neither? Show your math.

(b) Suppose instead for the same situation as (a) the expert says the probability it is a Russian submarine is 1.0. What evidence combination method is the expert using: independence-assumption, conservative, liberal, or neither? Show your math.

I-25 (for Chapter 8). Consider:

c :- a, b. c :- not(a), not(b). d :- a, not(b). e :- not(a), b.

(a) Suppose now all the above predicate expressions are given probabilities. Suppose further that the second rule has a rule strength of 0.8 and the third rule has a rule strength of 0.5. Write the modified rules containing probabilities.

(b) Suppose b has probability 0.4. Suppose we combine probabilities (including those for rule strengths) with the independence assumption. Conduct computer experiments to graph c, d, and e as a function of the probability of a. Use the representative a-probability values 0.0, 0.2, 0.4, 0.6, 0.8, and 1.0.

(c) Suppose b has probability 0.9. Suppose we combine probabilities (including those for rule strengths) with the independence assumption. Conduct computer experiments to graph c, d, and e as a function of the probability of a. Use the representative a-probability values 0.0, 0.2, 0.4, 0.6, 0.8, and 1.0.

(d) Suppose b has probability 0.4. Suppose we combine probabilities (including those for rule strengths) with the conservative assumption. Conduct computer experiments to graph c, d, and e as a function of the probability of a. Use the representative a-probability values 0.0, 0.2, 0.4, 0.6, 0.8, and 1.0.

(e) Suppose b has probability 0.9. Suppose we combine probabilities (including those for rule strengths) with the conservative assumption. Conduct computer experiments to graph c, d, and e as a function of the probability of a. Use the representative a-probability values 0.0, 0.2, 0.4, 0.6, 0.8, and 1.0.

(f) What major disadvantage of the conservative formulas is shown by the graphs for parts (d) and (e)? Note that being equal to zero a lot is not a disadvantage per se, because maybe the combination value really is zero.

I-26. (for Chapter 9) Suppose in a search you are in a state "a" and you can go to states "b", "c", and "d". Suppose "a" has evaluation 9, "b" 14, "c" 11, and "d" 15. Suppose the cost from "a" to "b" is 10, from "a" to "c" 20, from "a" to "d" 12, from "b" to "c" 15, and from "b" to "d" 8.

(a) What state do you go to next from "a" in a best-first search?

(b) What state do you go to next from "a" in an A* search?

(c) What state do you go to next from "a" in a depth-first search with the heuristic that states are preferred in alphabetical order?

I-27 (for Chapter 9). Consider as search the problem of using inheritance to find a value of a property for some object (Problem 9-2 and its answer may help).

(a) Would bidirectional search (in general) be a good idea?

(b) In which direction(s) would you expect the branching factors to be less?

(c) Give a domain-independent heuristic--that is, one that potentially applies to any inheritance situation, and doesn't reference features of the objects over which the inheritance occurs.

(d) Give a domain-dependent heuristic, one that depends on features of the objects over which inheritance occurs.

(e) Is the search (in general) decomposable? Why?

(f) Is the search (in general) monotonic for the operators "follow a_kind_of link", "follow part_of link", etc.? Why?

I-28. (for Chapter 10) Implement forward chaining as a search problem. Assume there are "rule" and "fact" facts defining the initial rules and facts. Give starting, "successor", and "eval" definitions. Assume no "not"s or other built-in predicates are used in the rules.

I-29. (for Chapter 10) Implement bidirectional best-first search by modifying the best-first program of the chapter. Alternate between forward and backward steps, where each step finds all the successors of a state. Assume you have also a backwards evaluation function, "backeval", and well as the usual "eval".

I-30. (for Chapter 11) Means-ends analysis can also involve numbers. To do this, the means-ends program can be modified to call on a user-defined predicate "transform" after the postconditions have been applied, but before the second recursive call. This "transform" has three arguments: an input operator name, an input state, and an output state representing the result state after necessary arithmetic calculations and adjustments have been done. For instance:

```
transform(smash_case,L,[value(0)|L2]) :- singlemember(value(X),L), delete(value(X),L,L2).
```

says that the "smash_case" operator changes the value of the flashlight from X cents to 0 cents; this would be in addition to all the preconditions, deletepostconditions, and addpostconditions shown in chapter 11. (The "singlemember" is a version of "member" that doesn't backtrack, which is already included in the means-ends program file.) To call on "transform", substitute for the "union" line in the means_ends definition the following:

```
union(Addpostlist,Prestate2,Postlist2), transform(Operator,Postlist2,Postlist),
```

Suppose you want to fight a fire in a compartment on board a ship, using means-ends analysis. Suppose you have three operators:

"hose": apply water from a hose to the fire for 10 seconds; "drain": open the drain; "wait": wait for the water to drain.

Suppose states can be described by a lists of the facts:

--fire(X): the fire is of degree X, where X is an integer, and larger X means a worse fire --open: the drain is open --closed: the drain is closed --flooded: the compartment is flooded --dry: the compartment is not flooded

And suppose these have as preconditions and postconditions (and nothing else is a precondition or postcondition):

--hose: you can't hose if the compartment is flooded, and you can't hose if the drain isn't open; each "hose" operation reduces the degree of the fire by 2, and floods the compartment.

--drain: you can do this anytime, and the drain stays open once opened.

--wait: you can do this anytime; if the compartment is flooded, it becomes unflooded; and the fire degree increases by 1 if it's not zero.

Assume in your starting state that the fire is degree 3, the drain is closed, and the compartment is dry. Your goal is a dry compartment with the fire out.

I-31. (for Chapter 11) Study problem 10-9. You are to program a computer in Prolog to solve the problem described, but using means-ends analysis (from Chapter 11) rather than A* search. Try to order the operator recommendations, preconditions, and postconditions in a reasonable way. Show your program working.

To simplify the problem:

--change the goal state to be one in which the robot must be in office 1.

--assume the robot can carry more than one basket at a time.

--assume it doesn't make sense to pick up a basket that isn't full.

--assume it doesn't make sense to put down a basket that is full.

--assume that if the robot is holding something, it carries it wherever it goes;

--don't reason about the last action taken in the recommended facts or the precondition facts.

--don't worry about the costs; don't worry if your answer seems a little stupid in doing some unnecessary things.

The choice of operators is important. Probably you'll need separate operators for picking up a trash basket, putting down the basket, carrying a basket somewhere, and disposing of the basket down the chute. Also distinguish carrying the basket from just going between rooms without carrying anything, because the two have different side effects. However, it would be poor programming to have two separate operators (and facts) for going from office 1 to the chute, and from the chute to office 1--except in Turbo Prolog, where you won't be able to use variables with operators (with some exceptions), and you'll have to write out each possible action as a separate fact.

You don't need any rules for recommendations, preconditions, and postconditions: you should be able to define them with just facts. The order of these facts makes a lot of difference. Also, the order of terms within the lists makes some difference too. Here's an example of an operator definition: the preconditions for carrying the basket that belongs in room X to place Y are that the robot is holding that basket, the robot is in room Z, and the basket is also in room Z. Then the deletepostconditions are that the basket is in room Z and the robot is in room Z. And the addpostconditions are that the basket is in room Y and the robot is in room Y. (The requirement that Y be different from Z can be enforced by the "recommended" facts.)

Since you can't write "negative preconditions" for the means-ends program, you'll need to define some "negative predicates" like "vacuumed" and "emptybasket" that can be used in preconditions.

Warning: be very careful you don't write your own definitions of the predicates difference, append, union, subset, deleteitems, and delete. Otherwise, your program will take forever to run, because of the unnecessary extra backtracking choices.

You'll need some debugging facilities, because the program you'll write will do some complicated things. It's particularly helpful to watch the calls to the "means_ends" predicate, because that tells you the identity of the overall goal the program is working on. One good way to debug is to check whether the arguments to means_ends make sense. Check for duplicate facts; these suggest mistakes in the deletepostconditions. Check for contradictory facts; these suggest mistakes in either the addpostconditions or deletepostconditions. Check for missing facts (for instance, if there are no facts at all about the dustiness of room 1); these suggest mistakes in the addpostconditions. When you've found a problem in the trace, try to find where it first occurs; the action then applied must have faulty definitions.

Another bug you may see is running out of stack space because of an infinite loop. Often this is due to the operator-defining facts being out of order. Other times it may be due to a recommendation or precondition failure of an operator.

I-32. (for Chapter 11) Modify the means-ends program to allow for "negative preconditions", things that must be false before an operator can be applied. These can mean simpler state descriptions. Implement them by facts in the preconditions with "not" in front of them, like saying "not(open(case))" instead of "closed(case)".

I-33. (for Chapter 13) Solve the following Prolog query by relaxation:

?- a(X,Y), a(Y,Z), a(X,Z), a(Z,Y).

given the database:

a(1,4). a(2,4). a(1,5). a(3,3). a(3,6). a(5,4).

I-34. (for Chapter 13) The scheduling program in section 13.1 is still very slow because duplicates aren't caught early enough. Modify the program to fix this.

I-35. (for Miscellaneous problems) (a) Which of the following is true (that is, most false) about writing artificial intelligence applications?

(i) Programs consist of many small pieces. (ii) Programmers implement algorithms. (iii) Programs use symbols more than numbers. (iv) Programmers spend much time debugging.

(b) Which of the following is true (that is, most false) about Prolog?

(i) It emphasizes "how" instead of "what".

(ii) It's usually interpreted, not compiled.

(iii) It's often good for combinatorial problems.

(iv) It's a relatively new language and can be developed further.

I-36. (for Miscellaneous problems) Linked lists are important in many ways in artificial intelligence. Give an important use of linked lists for each of the following.

(a) relaxation

(b) rule-based expert systems with probabilities

(c) breadth-first search, besides the state representation

(d) automatic backtracking

I-37. (for Miscellaneous problems) Which of the following techniques (not necessarily just one) can get into an infinite loop on some practical problems having solutions? For each answer "yes", explain how in a few words. (Assume there are no infinite processing loops in any problem-dependent code.)

(a) depth-first search with heuristics and a finite branching factor

(b) A* search with an always-positive cost function and a finite branching factor

(c) means-ends analysis

(d) rule-cycle hybrid chaining on rules without arguments

(e) inheritance of a property with respect to "part_of", for a finite set of facts

(f) inheritance of a property with respect to "is_another_name_for", for a finite set of facts

(g) proving something by resolution on clauses without arguments

I-38. (for Miscellaneous problems) Consider the data-flow diagram on the next page. The circles are data objects, and the rectangles are processes operating on the data objects.

.PA .PA

(a) Represent the entire diagram as a set of Prolog facts in the form

c(<input-object>,<process>,<output-object>).

where "c" stands for "connection". Some processes take two inputs; represent these with two facts. (Remember, "<" and ">" are not Prolog symbols, but a way to describe something.)

(b) Suppose that for some objects and some processes, we have additional facts of the form

defective(<object>). faulty(<process>).

An object is defective if it is the output of a process that is faulty. An object is also defective if it is the output of a process for which at least one input is defective. Write these as two Prolog inference rules that use the "c" facts. ("Defective" and "faulty" don't necessarily mean all the time; that is, the object or process could be correct part of the time. But you don't need to use probabilities in your rules.)

(c) Suppose process p is faulty; that is, we have the fact

faulty(p).

Explain how a Prolog interpreter would prove that object d is defective, using the facts from part (a) and the rules from part (b), and no additional rules. Give each step that the interpreter would take, including every variable binding. And give the starting query.

(d) Write the rules of part (b) in clause form.

(e) Prove by resolution that if object f is not defective, then object c is not defective either. Use the clauses from (d) and the facts from (a). (Warning: don't ever resolve clauses with variables in common; rename the variables first.)

(f) Suppose that in some data-flow diagram (not necessarily that in parts (a)-(e)) the objects X and Y are defective. Consider the problem of determining whether defective object X could explain defective object Y; that is, whether there's some path from X to Y. This can be considered a search problem in which the states are objects, the starting state is X, and the goal state is Y. If the branching factor is the same in both directions, would bidirectional search be a good idea? Why?

(g) Assume we are searching left-to-right with the diagram considered in parts (a) through (e). Assume we want to find a connection between an X and Y picked randomly, with a pick of any object in the diagram equally likely. What is the average branching factor for states (including state f)?

(h) Suppose we want a Prolog program to do this search. To use the search programs in the book, we must define a "successor" predicate of two arguments, an input state and an immediate successor output state.

Write its Prolog definition, using the "c" predicate of part (a).

(i) Now suppose we want to find all the defective objects and faulty processes, given a partial set of "defective" and "faulty" facts and the inference rules of part (b). Explain how relaxation could do this. What would be the variables? What would be the values? What would be the constraints?

(j) Suppose we know many facts about the processes (rectangles) in a data-flow diagram; then frames could represent processes. Consider the superframe (generalization frame) representing all occurrences of the "q" process in a data-flow diagram. Give examples (one each) of value and slot inheritance from this frame to a more specialized frame.

(k) Frames could represent sections of a data-flow diagram; these would be "part_of" superframes. For instance, the example diagram for part (a) could be considered one big process with "a" as input and "f" as output. This leads to hierarchical data-flow diagrams. Suppose for some process in a diagram there was an multiple-inheritance conflict between the "core-memory-required" value inherited from this "part_of" superframe and the "core-memory-required" value inherited from the generalization superframe of part (j). How would you resolve the conflict in a general and efficient way?

(l) Explain how to get a helpful lower-bound evaluation function for the search problem described in part (d), one that will work for any X and Y on any diagram. Assume the diagram is described by Prolog facts like those in part (a). To make your evaluation function work, you can store extra facts with each node of the diagram (but you can't select the facts based on knowledge of X and Y--the same facts must be used for every X and Y). Your evaluation function shouldn't actually solve the problem because then it would be a solution method, not an evaluation function.

.PA

ANSWERS TO MORE TEXT PROBLEMS

(This, and Appendix G in the book, cover the answers to nearly all the problems. The only exceptions are the problems that require very long answers.)

2-3. See the next page.

2-4. One way:

circle(c1). circle(c2). circle(c3). circle(c4). big(c1). small(c2). small(c3). small(c4). inside_of(c2,c1). right_of(c3,c1). above(c4,c3). touching(c1,c3). touching(c3,c4). same_center(c2,c1).

Note it's important to specify the type of c1, c2, c3, and c4 as circles, and important to specify "touching" relationships in addition to "right_of" relationships. Additional redundant facts could also be included beyond the preceding.

.PA .PA

2-8. Facts about the crime and the nature of the crime, how it was committed, and what objects were involved in it. These things don't require any common arguments except perhaps a specific code distinguishing the specific crime. Reports from witnesses can also include this code, but they will also all require two additional arguments: the name of the witness, and the time the specific observation was made. It's important to record the time, because the same action may occur more than once, and it's important to record the name of the witnesses because some witnesses lie, and their reports cannot be taken as facts. (By "time" we mean a unique identification for every unique time, so this may require a date as well as hours, minutes and seconds.)

2-9. Both can be expressed with type predicates, with predicate name "mayor". But "Clint" is a man's name and must represent a specific entity; "someone" is an English pronoun that can stand for any person. In fact, "someone" doesn't mean anything here because any "mayor" is a person. So "someone" is like a placeholder or "variable", to which a word can later be assigned. In particular, it's like an existentially quantified variable (see Appendix A). In Prolog, existentially quantified facts like that cannot be represented directly.

2-10. In the first sentence, "boss" indicates a relationship-predicate fact about two people; "boss" is like the predicate name. In the second sentence, it represents an action of ordering people around, and could be either a predicate name or an argument. (A "boss" does not necessarily "boss" people around, and not only "bosses" boss people around.) In the third sentence it refers to a symbol in English used to represent a word, so it's a statement about the symbols in a language; the word would be represented as an argument, not a predicate name. In the fourth sentence it refers to an abstract concept of being a boss, something related to but distinct from the bossing relationship in the first sentence; and the concept would be an argument, not a predicate. These four meanings are so different it would be poor judgment to use the same word for any of them in a computer representation, in line with the suggestion #3 in section 2.4 to avoid names with multiple meanings.

2-11. (a) It's more "natural", more like how people talk. So syntax errors are less likely than with predicates.

(b) There are many different ways of expressing things in English, and you must handle all of them.

(c) English is a lot more ambiguous than predicates. The word "is" is used in many different ways, as well as many other words like "the" and "at". So to make such an approach work, systems have to restrict the English input from the user. These restrictions are usually hard to remember and easy to get confused about.

3-3. The answers to query 2 are a subset or the same as the answers to query 1. Query 2 just "filters out" some of the answers to Query 1 by its additional predicates.

3-4. "The" signals a reference to something used before, unlike "a" and "an"; "the memo" refers to the previous "a memo", and "the meeting" to the previous "a meeting". So it's like the way variables are handled in a query, the "generate-and-test" approach: "the" corresponds to a bound variable, and "a" to an unbound variable.

3-7. (a)

```
married(jason,phoebe). married(phoebe,jason). married(perry,stacey). married(stacey,perry).
married(eulalie,lane). married(lane,eulalie). loves(phoebe,perry). loves(zack,phoebe). loves(lane,phoebe).
loves(jason,eulalie).
```

(b) Jason and Phoebe only, the only A and B values found below:

```
?- married(A,B), loves(A,C), loves(B,D), not(loves(A,B)), not(loves(B,A)). A=jason B=phoebe C=eulalie
D=perry ;

A=phoebe B=jason C=perry D=eulalie ;

no
```

(c) The jealous people are Phoebe, Zack, Lane, Jason, and Stacey. They are the possible A values in the following:

```
?- (loves(A,B); married(A,B)), loves(C,B).

A=phoebe B=perry C=phoebe ;

A=zack B=phoebe C=zack ;

A=zack B=phoebe C=lane ;

A=lane B=phoebe C=zack ;

A=lane B=phoebe C=lane

A=jason B=eulalie C=jason ;

A=jason B=phoebe C=zack ;

A=jason B=phoebe C=lane ;
```

```
A=stacey B=perry C=phoebe ;

A=lane B=eulalie C=jason ;

no
```

3-8.(a) The query asks for "a" facts whose arguments match those of "b" facts. So there are two answers: X=3 and Y=2, and X=4 and Y=4.

(b) This query imposes an additional restriction on the solutions to part (a); X=4 and Y=4 is the only possibility that satisfies the additional predicate expression.

3-10.(a) Twice: once for X=a and Y=b, and once for X=b and Y=b. The binding X=c and Y=d succeeds.

(b) Three times: once for X=a, Y=b, and Z=b; once for X=b, Y=b, and Z=b; and once for X=c, Y=d, and Z=a.

(c) None--Prolog interpreters don't backtrack to the last query predicate unless a semicolon is typed.

(d) Once: for the last binding X=c and Y=d. Then the binding X=c and Y=a is found and it succeeds.

(e) Once: for the last binding X=c, Y=d, and Z=a.

3-11. One way:

subdepartment(<department>,<subdepartment>). project(<project-name>,<project-code>, <starting-date>,<completion-date>,<subdepartments>). employee(<employee-name>,<ss-number>,<birth-date>, <home-address>,<office>,<job-skills>). employee_project_match(<ss-number>,<project-code>).

Note that since usually a project has only one subdepartment, we might as well make that attribute an argument in the "project" facts; but since the number of employees in a project could vary widely, the association of employees to projects should be handled by separate "employee_project_match" facts in which both the same employee and the same project can be mentioned multiple times. Note that job skills can go in the "employee" facts since they can efficiently be stored by a bit array, giving a fixed record length for all employees.

3-12. (a) The computer took the question too literally; it should have realized that what the person really wanted to know was the name of that commander and his rank, not whether it had the capability of finding out those things. So a natural language front end must have expectations about what it will be called upon to do, and it must live up to those expectations rather than answering a question literally.

(b) Again the computer is taking the user's questions too literally, but now the bug is more subtle because the answers given are correct for sailboats and other small ships that don't have recorded locations and commanders. In order to answer the question, the computer discovered that the Pequod does not exist, and then it concluded that there are never any commanders for ships that do not exist. But that first conclusion is much more interesting than the second, so the computer should have given it instead. That feature of computer behavior is called being "cooperative".

4-1. For one thing, the ":-" has a procedural meaning, not just a declarative one like logical implication. For another, the ":-" also includes a causal relation between the thing implying and the thing implied. A logical implication has no such causal meaning, and can in fact be equally well used in a backwards or "contrapositive" way, whereby if A implies B, then not(B) implies not(A). Chapter 14 discusses this more.

4-3. One way:

```
father(X,Y) :- child(X,M,Y,S).
```

```
mother(X,Y) :- child(F,X,Y,S).
```

```
son(X,Y) :- child(Y,M,X,male). son(X,Y) :- child(F,Y,X,male).
```

```
grandfather(X,Y) :- father(X,Z), parent(Z,Y). parent(X,Y) :- father(X,Y). parent(X,Y) :- mother(X,Y).
```

```
sister(X,Y) :- child(F,M,X,female), child(F,M,Y,S), not(same(X,Y)). same(X,X).
```

```
uncle(X,Y) :- parent(P,Y), brother(X,P). uncle(X,Y) :- parent(P,Y), sister(Z,P), married(Z,X).
brother(X,Y) :- child(F,M,X,male), child(F,M,Y,S), not(same(X,Y)). married(X,Y) :- child(X,Y,C,S).
married(X,Y) :- child(Y,X,C,S).
```

```
ancestor(X,Y) :- parent(X,Y). ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).
```

```
halfsister(X,Y) :- child(F,M,X,female), parent(P,X), parent(P,Y), not(sister(X,Y)), not(same(X,Y)).
```

4-4. Code:

```
on(b,a). on(c,b). on(d,c). on(f,e). on(g,f). on(i,h). left(a,e). left(e,h). above(X,Y) :- on(X,Y). above(X,Y) :-
on(X,Z), above(Z,Y). stackleft(X,Y) :- left(X,Y). stackleft(X,Y) :- left(X,W), above(Y,W). stackleft(X,Y)
:- left(Z,Y), above(Z,X). stackleft(X,Y) :- left(Z,W), above(Z,X), above(Y,W).
```

Run:

```
?- on(X,b).
```

```
X=c ;

no ?- on(a,X).

no ?- on(X,Y).

X=b Y=a ;

X=c Y=b ;

X=d Y=c ;

X=f Y=e ;

X=g Y=f ;

X=i Y=h ;

no ?- on(X,Y),left(Y,Z).

X=b Y=a Z=e ;

X=f Y=e Z=h ;

no ?- above(X,Y).

X=b Y=a ;

X=c Y=b ;

X=d Y=c ;

X=f Y=e ;

X=g Y=f ;

X=i Y=h ;

X=c Y=a ;
```

```
X=d Y=b ;

X=d Y=a ;

X=g Y=e ;

no ?- above(X,f); (left(Y,Z),above(X,Y),above(f,Z)).

X=g Y=_5 Z=_6 ;

X=b Y=a Z=e ;

X=c Y=a Z=e ;

X=d Y=a Z=e ;

no ?- above(X,Y), not(left(X,Z)).

X=b Y=a Z=_6 ;

X=c Y=b Z=_6 ;

X=d Y=c Z=_6 ;

X=f Y=e Z=_6 ;

X=g Y=f Z=_6 ;

X=i Y=h Z=_6 ;

X=c Y=a Z=_6 ;

X=d Y=b Z=_6 ;

X=d Y=a Z=_6 ;

X=g Y=e Z=_6 ;

no
```

4-8. X=3, Y=3. X=1 and Y=2 can't match either of the "b" facts or the "b" rule. X=3 and Y=5 can't match the "b" facts, and matching the "b" rule would require that "b(2,5)" be provable, and it isn't. So the "a" rule must be used if there is to be any answer to the query. R=1 and S=3 won't work because there isn't any "b(2,1)" fact; R=2 and S=3 won't work because there isn't any "b(2,2)" fact. So the "b" rule must be used to get an R and an S. But on its right side, T can be bound to 3 and match the two "b" facts. So "b(3,3) is proved, and hence "a(3,3)" is proved.

4-9. For the relationship "part_of", the properties "author", "language_written_in", and "source_institution" inherit downwards (that is, from the program to the subprogram); the property "has_bugs" inherits upwards. For the relationship "a_kind_of", the properties "has_applications_to", "requires_128K_machine", and "Company_X_sells_a_program_of_this type" inherit upwards (from more specific program to more general program).

4-12. (a) Upwards. Properties involving existential quantification usually inherit upwards.

(b) Neither direction. Statistical properties usually don't inherit either way.

(c) Downwards. Properties involving universal quantification usually inherit downwards.

(d) Downwards. Another universal quantification.

4-14. (a) 0--it could be a fact. (Or maybe 1 if you read the question as meaning the second transitivity rule must be used once.)

(b) Twenty-nine, when the objects all form a single chain. Note that the number of connections in a chain of number N items is N-1, and we can leap the final gap with a fact instead of a rule--so that's N-2 uses of the second transitivity rule.

(c) A balanced binary tree of height 4.

(d) 3, the height of the tree minus one, (since we can leap the last gap with a fact).

(e) Once, since the only recursion necessary is done by the transitive predicate "a_kind_of" and not in the inheritance rule itself.

4-15.(a) Note that Tom's Rabbit bears the same relationship to all Rabbits as all Rabbits do to all VWs: the first thing is a kind of the second thing. Similarly, an alternator bears the same relationship to an electrical system as an electrical system does to a VW: the first thing is part of the second thing. Note also that three English phrases represent single concepts and thus can each map to a single word in Prolog: VW Rabbit, Tom's car, and Dick's car; we use the underscore character (_) in the Prolog words in these situations. So we get (noting that no characters may be capitalized):

a_kind_of(vw_rabbit,vw). a_kind_of(toms_car,vw_rabbit). a_kind_of(dicks_car,vw_rabbit).

part_of(electrical_system,vw). part_of(alternator,electrical_system). defective(alternator).

The last fact does not need to be qualified as to the kind of car since we're only talking about VW's here. To use artificial intelligence terminology, VWs are the universe of discourse. You also might want to include facts of ownership, and the statement that the alternator is on the VW, though you don't need them for parts (b) and (c):

owns(tom,vw_rabbit). owns(dick,vw_rabbit). on(alternator,vw).

Note that our predicates are quite general--this is the usual way that artificial intelligence knowledge representation is done. The specific things mentioned like VW Rabbits, Tom's car, and electrical systems should be arguments to predicates, not the names of predicates. Then we don't have to write a lot of different inferences rules for all our predicate names. So all the following would be poor knowledge representation:

vw(vw_rabbit). vw_rabbit(tom). vw_part_of(electrical_system). has_electrical_system(vw). alternator_defective(vw).

Also note that nothing is asserted about VWs being cars or VW Rabbits being cars--only that Tom has a car and Dick has a car.

(b) You need to say that a_kind_of and part_of are transitive:

a_kind_of(A,C) :- a_kind_of(A,B), a_kind_of(B,C). part_of(D,F) :- part_of(D,E), part_of(E,F).

and also the defectiveness inherits with respect to them. That is, that something is defective if it is a member of a class of things that are defective, or if any part of it is defective:

defective(G) :- a_kind_of(G,H), defective(H). defective(I) :- part_of(J,I), defective(J).

Note the reversed order of arguments in the second rule--defectiveness inherits upwards for part_of, and downwards for a_kind_of.

Alternatively, you could use the on(alternator,vw) information with this rule:

part_of(K,L) :- on(K,L)

and eliminate the second rule above. But this is controversial--something could be "on" a car, like a ski rack, and not be part of the car.

(c) The goal is to prove defective(dicks_car). By the fourth rule above, it is sufficient to show the some part of Dick's car is defective. Using the part_of transitivity rule (the second rule above) with

D=alternator, E=vw, F=electrical_system, or the last rule above, we see that the alternator is part of the VW. Since all alternators are defective, it follows all VWs are defective. Now using the first rule with A=dicks_car, B=vw, C=vw_rabbit, we see that Dick's car is a VW. Hence by the third rule, Dick's car is defective.

4-16. One Prolog database:

limit(15) :- near_railroad, railroad_view_blocked, not(railroad_warning). railroad_warning :- railroad_gates. railroad_warning :- railroad_signal. railroad_warning :- flagman. limit(15) :- near_intersection, intersection_view_blocked, not(right_of_way). right_of_way :- cross_street_yield. right_of_way :- cross_street_stop. limit(15) :- in_alley. limit(X) :- sign(limit,X), X<25. limit(25) :- sign(school), children_danger. children_danger :- children_present. children_danger :- unfenced-schoolyard. limit(X) :- sign(limit,X), X<56. limit(25) :- district(residential). limit(55).

First database:

sign(limit,55). district(residential).

Second database:

sign(school). children_present. sign(limit,55). district(residential).

Third database:

sign(limit,35). in_alley. sign(school). children_present. sign(limit,55). district(residential).

With the first situation:

?- limit(X).

X=55

yes

With the second situation:

?- limit(X).

X=25

yes

With the third situation:

?- limit(X).

X=15

yes

4-17. Legal definitions are much more flexible, because they deal with generally much more abstract notions. It's not a weakness of either legal definitions or Prolog it's just that they deal with different things. It probably wouldn't be a good idea to make legal definitions more precise, because then they couldn't be applied to all the cases they need to be applied to. It probably wouldn't be a good idea to make Prolog less precise because precision is why people use computers in the first place.

4-18. Often the terms discussed are not clearly defined, and mean different things to different people. So people are mapping the same sentences into different internal representations. And because such terms are often abstract, it's hard for people to understand the different meanings that each is assigning.

4-19. (a) A piece of the rock.

(b) Chemical decomposition of the rock into silicon and oxygen molecules.

4-20. This is the famous "barber" paradox of Bertrand Russell. (a) saves(X,God) :- not(saves(X,X)).

(b) The rule becomes: saves(God,God) :- not(saves(God,God)).

(c) No, the proper conclusion is that the original rule is faulty. Just because we can write something with Prolog notation doesn't make it true.

5-3.

better_divide(X,Y,Q) :- not(var(X)), not(var(Y)), not(var(Q)), not(Y=0), Z is X / Y, Z=Q. better_divide(X,Y,Q) :- not(var(X)), not(var(Y)), var(Q), not(Y=0), Q is X / Y. better_divide(X,Y,Q) :- not(var(X)), var(Y), not(var(Q)), Y is Q * X. better_divide(X,Y,Q) :- var(X), not(var(Y)), not(var(Q)), X is Q * Y.

5-4. Use "better_add" and:

tax(G,D,R,T) :- better_add(D,N,G), better_multiply(N,R,T). tax(G,D,R,T) :- better_multiply(N,R,T), better_add(D,N,G). better_multiply(X,Y,S) :- not(var(X)), not(var(Y)), not(var(S)), Z is X * Y, Z=S. better_multiply(X,Y,S) :- not(var(X)), not(var(Y)), var(S), S is X * Y. better_multiply(X,Y,S) :-

```
not(var(X)), var(Y), not(var(S)), not(X=0), Y is S / X. better_multiply(X,Y,S) :- var(X), not(var(Y)),
not(var(S)), not(Y=0), X is S / Y.
```

Notice no "var" predicate expressions are needed in the "tax" rules; if more than one argument to "tax" is unbound, both "better_add" and "better_multiply" will fail.

5-5.

```
square(X,Y) :- not(var(X)), not(var(Y)), Z is X*X, Z=Y. square(X,Y) :- not(var(X)), var(Y), Y is X*X.
square(X,Y) :- var(X), not(var(Y)), square_bisection(X,Y,0,Y). square(X,Y) :- var(X), var(Y),
generate_pair(X,Y).
```

```
square_bisection(X,Y,Lo,Hi) :- X is (Lo+Hi)/2.0, square(X,S), close(S,Y). square_bisection(X,Y,Lo,Hi) :-
Midpoint is (Lo+Hi)/2.0, square(Midpoint,S), S<Y, square_bisection(X,Y,Midpoint,Hi).
square_bisection(X,Y,Lo,Hi) :- Midpoint is (Lo+Hi)/2.0, square(Midpoint,S), S>=Y,
square_bisection(X,Y,Lo,Midpoint).
```

```
close(X,Y) :- D is X-Y, D > -0.001, D < 0.001.
```

```
generate_pair(1,1). generate_pair(X,Y) :- generate_pair(A,B), X is A+1, Y is (A+1)*(A+1).
```

5-6. (a) Use the four lines defining "better_add", but change the name to "integer_better_add" and add one additional line:

```
integer_better_add(X,Y,S) :- var(X), var(Y), not(var(S)), integer_in_range(X,1,S), Y is S - X.
```

where "integer_in_range" is defined as in the chapter.

(b) Let S be the characteristic sum for the magic square, and let the letters of the alphabet starting with A represent the cells in the magic square in row-major order. Then:

```
magic_square(S,A,B,C,D,E,F,G,H,I) :- add_3(A,B,C,S), add_3(A,D,G,S), add_3(D,E,F,S),
add_3(B,E,H,S), add_3(G,H,I,S), add_3(C,F,I,S), add_3(A,E,I,S), add_3(C,E,G,S). add_3(X,Y,Z,S) :-
integer_better_add(X,W,S), integer_better_add(Y,Z,W).
```

Note that the order of the expressions in the first rule is very important to the efficiency.

5-7. (a) X=3, Y=2. Notice the query contains three terms "and"ed together. The first two predicate-a facts fail to satisfy the second condition in the query because c(0) is a fact. The third a fact fails because of the d condition. So we must use the rule for a. The first condition matches with P=3 and Q=1, but the second condition fails. So P=2 and Q=1 is tried and a(3,2) succeeds. The "not" condition and the d(3,2) succeed so the whole query succeeds, and X=3 and Y= 2.

(b) X=2, Y=3. Obtained by the next possible match for b(P,Q), that of P=1 and Q=2. This succeeds with the other two query terms similarly.

5-9.(a) speed(<gear_number>,<speed_in_rpm>). Positive speeds can be clockwise, negative counterclockwise.

(b) speed(G,S) :- same_shaft(G,G2), speed(G2,S).

(c) speed(G,S) :- meshed(G,G2), teeth(G,TG), teeth(G2,TG2), speed(G2,S2), S is 0 - (S2 * TG2 / TG).

(d) Use the following database:

speed(g1,5000). same_shaft(g2,g1). meshed(g3,g2). meshed(g4,g2). teeth(g1,100). teeth(g2,30). teeth(g3,60). teeth(g4,90). speed(G,S) :- same_shaft(G,G2), speed(G2,S). speed(G,S) :- meshed(G,G2), teeth(G,TG), teeth(G2,TG2), speed(G2,S2), S is 0 - (S2 * TG2 / TG).

and issue this query:

?- speed(g4,S).

The one speed fact doesn't apply, so the first speed rule is tried. But there is no same_shaft fact with g4 as first argument, so the rule fails. Now the second speed rule is tried. G2 can be matched to g2, with TG=90 and TG2=30. Then we must find the speed of G2=g2. No fact applies, but we can use the first speed rule: g2 is the first argument to a same_shaft fact having g1 as second argument, so its speed is the speed of g1. And a fact says the speed of g1 is 5000. So the speed of g2 is 5000, and we can perform the calculation in the second speed rule as

S = 0 - (5000 * 30 / 90) = -1667 rpm

and the variable S in the original query is bound to that number.

(e) You could get infinite loops if you had extra redundant facts, like if you had both meshed(g2,g1) and meshed(g1,g2). You could also get infinite loops if you had more than one inference rule of each of the two above types, as if you had both

speed(G,S) :- same_shaft(G,G2), speed(G2,S). speed(G,S) :- same_shaft(G2,G), speed(G2,S).

(f) The gears wouldn't turn--such a contradiction of rotation speeds is an impossibility.

5-10. It's easiest to just convert everything to meters, since you must worry about infinite loops otherwise.

```
convert(M1,U1,M2,U2) :- conversion_factor(meters,U1,F1), conversion_factor(meters,U2,F2), M2 is
M1*F2/F1.
```

```
conversion_factor(meters,meters,1). conversion_factor(meters,decimeters,10).
conversion_factor(meters,centimeters,100). conversion_factor(meters,millimeters,1000).
conversion_factor(meters,decameters,0.1). conversion_factor(meters,kilometers,0.001).
conversion_factor(meters,inches,39.37). conversion_factor(meters,feet,3.281).
conversion_factor(meters,yards,1.094). conversion_factor(meters,miles,0.0006214).
```

```
?- convert(2,feet,X,meters).

X=0.609571

yes ?- convert(2,feet,X,inches).

X=23.9988

yes ?- convert(20,kilometers,X,miles).

X=12.428

yes ?- convert(42.1,yards,X,decameters).

X=3.84826

yes
```

5-11.(a)

```
route(Town1,Town2,Routename,County,Distance), stateof(County,State), countryof(State,Country),
limit(State,Limit), limit(Country,Limit).
```

(b)

```
limit(R,L) :- route(T1,T2,R,C,D), stateof(C,S),limit(S,L). limit(R,L) :- route(T1,T2,R,C,D),
stateof(C,S),countryof(S,K), limit(K,L).
```

(c)

```
distance(T1,T2,D) :- route(T1,T2,R,C,D). distance(T1,T2,D) :- route(T1,T3,R,C,D3),
distance(T3,T2,D4), D is D3+D4.
```

(d) r+c facts

(e) infinity, (if routes car contain same town many times). Otherwise, between 0 and t*t - t - r.

Alternative: assign county to town

(a) route(Town1,Town2,Routename,Dist), countyof(Town,County), stateof(County,State), countryof(State,Country), limit(CountyorState,Limit)

(b) limit(R,L) :- route(T1,T2,R,D), countyof(T1,C), countyof(T2,C), stateof(C,S), limit2(S,L). limit2(S,L) :- limit(S,L). limit2(S,L) :- countryof(S,K),limit(K,L).

(c) distance(T1,T2,D) :- route(Tl,T2,R,D). distance(T1,T2,D) :- route(T2,T1,R,D). distance(T1,T2,D) :- route(T1,T3,R13,D13), distance(T3,T2,D32), D is D13 + D32.

5-12. Use:

inference_distance(X,X,0). inference_distance(X,Y,N) :- a_kind_of(X,Z), inference_distance(Z,Y,N2), better_add(N2,1,N).

where "better_add" is defined in the chapter. For a demonstration, assume we have these facts in our database:

a_kind_of(a,c). a_kind_of(b,c). a_kind_of(c,d). a_kind_of(c,e).

Here is a demonstration, showing the testing of all eight cases. (It's important to test all of them, because even a simple Prolog program can surprise you.)

?- inference_distance(a,c,1).

yes ?- inference_distance(a,c,N).

N=1 ;

no ?- inference_distance(a,Y,2).

Y=d ;

Y=e ;

```
no ?- inference_distance(X,d,2).

X=a ;

X=b ;

no

?- inference_distance(X,Y,2).

X=a Y=d ;

X=a Y=e ;

X=b Y=d ;

X=b Y=e ;

no ?- inference_distance(X,c,N).

X=c N=0 ;

X=a N=1 ;

X=b N=1 ;

no ?-inference_distance(a,Y,N).

Y=a N=0 ;

Y=c N=1 ;

Y=d N=2 ;

Y=e N=2 ;

no ?- inference_distance(X,Y,N).

X=_0 Y=_0 N=0 ;

X=a Y=c N=1 ;
```

X=a Y=d N=2 ;

X=a Y=e N=2 ;

X=b Y=c N=1 ;

X=b Y=d N=2 ;

X=b Y=e N=2 ;

X=c Y=d N=1 ;

X=c Y=e N=1 ;

no

5-13. The function cannot be a strong homomorphism. That is, it cannot map a lot different situations onto a small set of values. If so, odds are it is difficult, or perhaps even impossible, to fill in other arguments given the value of the last argument and perhaps a few others. Notice that much of arithmetic involves strong homomorphisms: there are many ways to add numbers to get the same sum or the same product. So it is understandable that the Prolog "is" usually used with arithmetic isn't reversible.

5-15. There are two different ways the Prolog interpreter can recurse in in this situation, on lists of increasing size for its first argument and on lists of increasing size for its second argument. It can only recurse one way at a time. So it picks the second way and only generates appending situations where the first argument is the empty list.

5-16. (a) It finds an item, if any, that it can delete from the second-argument list to create the third-argument list.

(b) It will delete only the first occurrence of the first argument from the second-argument list.

5-17. (a) It replaces every other word in a list with the word "censored", binding the result to the second argument. The rule (third line) just says if you can find an X and Y at the front of your list first argument, change the Y to the word "censored" after recursively doing the same on the rest of the list.

The easiest way to see this is figure out what happens with simple example lists:

?- mystery([],Z). Z=[] ?- mystery([a],Z). Z=[a] ?- mystery([a,b],Z). Z=[a,censored]

For the last example the third rule applies for the first time. So X is matched to a, Y is matched to b, and L is matched to [], the empty list. The rule involves a recursive call with first argument L, but L is the empty list, it's the same situation as the first example above, and M is bound to the empty list too. So the rule says to take M and X and the word "censored" and assemble them into the result, the list whose first two items are X and "censored", and whose remainder is M. But M is empty, so the result is [a,censored].

Now consider the next hardest case, an example list with three items:

```
?- mystery([a,b,c],Z).
```

The third rule must be again used. It matches X to a, Y to b, and L to [c], the list containing only c. The rule says to do a recursion with [c] the first argument. But to solve that the second rule will do, so M is bound to [c]. So the answer is assembled from X, the word "censored", and that M, and the result for Z is [a,censored,c].

Now consider an example list with four items:

```
?- mystery([a,b,c,d],L).
```

In the third rule, X is matched to a, Y is matched to b, and [c,d] is matched to L. The recursion applies mystery to the two-item list [c,d], which by analogy to the above two-item example gives [c,censored]. So we assembled the answer from X, the word "censored", and the list [c,censored], and the result for Z is [a,censored,c,censored]. So it looks like the program changes every alternate word in a list to the word "censored".

(b) One for even-length lists, one for odd-length. The rule lops off two items from the list on each recursion, so you can get into two different final situations.

5-18. See the next page.

.PA .PA

5-19. (a) N + 1 times, once for every item in the list plus once for the empty list; all these invocations will fail.

(b) (N-1)/2. There are 0 calls if the item is first in the list, one call if the item is second in the list, two calls if the item is third in the list, and so on. These terms form an arithmetic series. The average value of an arithmetic series is the average of the first and last numbers, in this case 0 and N - 1.

5-20. Write substitution rules with variables instead of substitution facts. The right sides of the rules can require that the variable bindings be drawn from lists of appropriate words, using the "member" predicate. For example:

```
substitution([X,transition],[X,change]) :-

member(X,[will,must,should,can,might,perhaps,always,never]).
```

5-21. The basis condition is wrong. The predicate only makes sense when you can compare horses in a set. For that, you need a basis step concerning at least two horses. But obviously two horses aren't necessarily the same color. Moral: even though basis conditions are usually simple, don't slight them in writing recursive programs: they can have some subtle bugs.

5-22. Use a list to keep arguments previously used, and just check to make sure a new value found is not in the list. At every recursion, add the new argument to the list. So:

```
a3(X,Y) :- a2(X,Y,[]). a2(X,Y,L) :- a(X,Y). a2(X,Y,L) :- not(member(X,L)), a(X,Z), a2(Z,Y,[X|L]).
member(X,[X|L]). member(X,[Y|L]) :- not(X=Y), member(X,L).
```

Then you query a3 to compute relationships by transitivity. For instance, suppose you have the database:

```
a(r,s). a(s,t). a(t,r).
```

(For instance, predicate "a" might mean "equals" or "is near".) Then the query

```
?- a(r,t).
```

will succeed as it should, as will

```
?- a(r,r).
```

But the query

```
?- a(r,u).
```

will fail as it should.

5-23. First define:

```
alldifferent([]). alldifferent([X|L]) :- not(member(X,L)), alldifferent(L).
```

Then define rhymes by facts with a list argument, for instance:

```
rhymes([ooga,booga,dooga,googa]). rhymes([itty,mitty,litty,sitty,ritty]).
rhymes([drallak,pallak,thrallak,chrallak]). rhymes([morade,brocade,hodade,volade]).
rhymes([fufu,rufu,nufu,pufu]). onerhymes([bo,go,toe,joe,moe]).
```

To pick combinations of two or three rhyming words, we need:

```
rhymes2(X,Y) :- rhymes(L), member(X,L), member(Y,L), alldifferent([X,Y]). rhymes2(X,Y,Z) :-
rhymes(L), member(X,L), member(Y,L), member(Z,L), alldifferent([X,Y,Z]). onerhymes2(X,Y,Z) :-
onerhymes(L), member(X,L), member(Y,L), member(Z,L), alldifferent([X,Y,Z]).
```

Finally we must define a predicate that puts the poem together:

```
poem([[A1,A2,B1],[C1,C2,B2],[D1,D2],[D3,D4],[E1,E2,B3]]) :- rhymes2(D1,D2), rhymes2(D3,D4),
alldifferent([D1,D2,D3,D4]), onerhymes2(B1,B2,B3), rhymes2(A1,A2),
alldifferent([A1,A2,D1,D2,D3,D4]), rhymes2(C1,C2), alldifferent([C1,C2,A1,A2,D1,D2,D3,D4]),
rhymes2(E1,E2), alldifferent([E1,E2,C1,C2,A1,A2,D1,D2,D3,D4]).
```

Then to write a poem we just query

```
?- poem(P).
```

6-2. The first fact data(3,0,1) can match the first predicate on the right side of the second rule, with A=3 and B=1, giving a new rule:

```
bottom(3,1,7,D) :- data(3,D,1).
```

But the same fact can also match this new rule too with D=0. So we can prove the fact bottom(3,1,7,0). We have exhausted the first fact, so we now consider this new fact. It can match the right side of the "top" rule, so a new fact top(0,7,3) is proved. We now consider the only remaining fact, data(3,2,1), the second of the original two facts. This can match the original "bottom" rule, giving:

```
bottom(A,B,7,2) :- data(A,0,B).
```

But data(3,2,1) can also match the first-listed rule above, giving a new fact bottom(3,1,7,2). This then matches the right side of the "top" rule giving the final new fact top(2,7,3).

6-3.(a) M*min(F,S) matchings. The number of facts bounds the number of predicate names involved, but so does S, so we must take the minimum.

(b) L facts. Even if F=1 there exist rule-based systems for which every conclusion can be reached.

(c) T matchings. If the right side predicates all have the same predicate name, they could all match a single fact.

(d) If all predicates are single-argument, each of the L distinct right sides could be bound in F+T-1 ways, if all F facts and T right-side expressions have the same predicate name, and there are T-1 distinct constants on right sides, giving L*(F+T-1) maximum conclusions.

6-4.(a) d(5), c(2), c(5), b(5), a(5). You cycle through the rules in order. The third rule succeeds first, then the fourth rule (which can be used in two different ways). Returning to the top of the list of rules on the second cycle, the second rule can now succeed. Then on the third cycle the first rule can succeed.

(b) c(2), c(5), d(5), b(5), a(5). The first fact is g(2) which can match the right side of the last rule, so c(2) is proved. This implies the new rule b(2) :- d(2). The second fact is f(5) which can match part of the third rule right side, giving d(5) :- e. The third fact is g(5) which can match the last rule, so c(5) is proved. This implies the new rule b(5) :- d(5) analogous to the earlier offspring of this rule. The last fact is e, which can be used to proved d(5) from our previously derived d rule. This new fact d(5) can match the right side of another derived rule, proving b(5). This matches the right side of the first rule, proving a(5).

6-6. The facts true at some point in hybrid chaining are like the object being worked on an assembly line. New facts represent new parts or new connections made to the objects on the assembly line. As the rules proceed a major conclusion is "built up" like an object assembly. Some rules may not succeed in producing new facts, and they are like optional equipment on the assembly line that is only used for production of certain kinds of objects.

6-10. (a) In decision lattices a single best path is taken through the lattice based on questions asked at each node. In the and-or-not all possible paths are explored in parallel (often by electronic signals). So decision lattices require user interaction, while and-or-not lattices don't. Decision lattices require a lot of work to construct from Prolog rules, while and-or-not lattices are a simple translation of what Prolog rules mean. Decision lattices can represent nearly any rule-base system, whereas and-or-not lattices can only represent rule-base systems with no or limited variables. Decision lattices are good when you want to save space because their implementation is very simple, whereas and-or-not are good when you want to save time, and especially good when you have access to special-purpose hardware.

(b) Both represent "compilation" of rule-base systems, more efficient but harder to understand and modify than Prolog rules describing the same rule-base system.

6-11.(a) One solution: assign a fixed rule order of H, I, J, M, K, L, D, C, E, F, B, G, A. Then with backwards chaining, rules would be invoked in order H, I, D, J, E, M, D, K, F, L, A. Note you don't actually have to do any work to satisfy C and E here.

(b) Forwards chaining and rule-cycle hybrid chaining would also work well. Another example: use a "use-same-room-if-possible" meta-rule as the basis of a control structure. That is, prefer to apply rules the mention the current room (and if there are still two rules that apply, chooses the first one in rule order). Then the above rules would be invoked in order H, I, D, J, E, M, D, K, F, L, A (same order as the preceding, but rules could be listed in any order).

6-13. (a) plaintiff_money(M) :- award(A), pool(P), M is A+25+(P/2). defendant_money(M) :- award(A), pool(P), M is 25+(P/2). pool(0) :- award(A), A>=500. pool(P) :- award(A), A<500, P is 500-A.

(b) plaintiff_money(M) :- award(A), A>=500, M is A+25. defendant_money(25) :- award(A), A>=500. plaintiff_money(M) :- award(A), A<500, M is 275+(A/2). defendant_money(M) :- award(A), A<500, M is 275-(A/2).

(c) Yes, a large rule-based system is needed to apply laws of mathematics to simplify the arithmetic expressions created by the substitution process, simply because there are so many laws of mathematics. So-called "symbolic algebra" systems can do this, and they're generally big, complex programs.

(d) You lose the ability to name parts of the computation, so it's harder to debug and modify such programs. Also, the mathematical expressions tend to get more complicated (though it didn't happen above), and it's harder to see what's happening.

(e) This is for compiling arithmetic, not logic. It doesn't map to hardware so easily. It does handle variable bindings.

6-14. (a) It's a meta-rule, a rule controlling the selection of other rules.

(b) "Simplest" is very hard to define. It can't just mean the rules with the fewest predicates on their right sides, because the complexity of those predicates matters even more. You can't just assign overall "complexity numbers" to those predicates either, because the complexity is a function of the context in which the predicates occur and the way they relate to each other as much as in the predicates themselves. Also, there's another problem: the simplest explanation of something is often the most general explanation, and such explanations aren't very useful.

6-15. Bureaucracies are less deterministic: if orders are given, they may not be followed, or may be misunderstood and misapplied when followed. Still, there are analogies. Policies, regulations, and orders are formulated to constrain the activities of subordinates in certain desirable ways. So they can eliminate the possibility of applying certain action methods or rules (like meta-rules), or they can set the priority of consideration of action methods or rules (like rule-ordering criteria).

6-16. Good names are important in any programming. So we'll define:

"above_limit"=S1, "at_limit"=S2, "below_limit"=S3, "two_lane"=S4, "four_lane"=S5, "intersection"=S6, "car_close"=S7, "2_to_4"=S8, "4_to_2"=S9, "brake_lights"=S10, "closing"=S11, "passing"=S12; and "speed_up"=A1, "slow_down"=A2, "maintain_speed"=A3, "pass"=A4. Conditions for slowing down are most critical, so their rules should go first. One possible set of rules:

action(slow_down) :- danger. action(slow_down) :- above_limit. danger :- car_close, brake_lights. danger :- 4_to_2. danger :- intersection. danger :- confused. action(speed_up) :- not(danger), below_limit. action(pass) :- passing. action(pass) :- not(4_to_2), not(intersection), car_close, closing. action(maintain_speed). confused :- 2_to_4, 4_to_2. confused :- above_limit, below_limit. confused :- above_limit, at_limit. confused :- at_limit, below_limit.

Many more rules defining "confused" can be added; they're error checkers. You query these rules by:

?- action(X).

much like the traffic-lights program of Chapter 4.

6-17. (a) Random changes to a system that always does completely thorough reasoning with a set of goals or facts just changes the order of that reasoning, not the set of conclusions reached. But if we assume people do incomplete reasoning much of the time--that is they don't follow their reasoning chains very far--then random rearrangement might make certain conclusions possible that weren't reasonable before. In fact, there's much evidence that people do have such limitations, tending to get lost in complicated reasoning chains. Of course, such random rearrangements could also make other inferences that used to be easy become harder.

(b) Again, if a system always does complete reasoning on a set of facts or a set of goals, a speedup of a factor of K in the reasoning for all the steps in a chain means a speedup of K for the entire chain. But if we assume that people don't do complete reasoning, and in fact tend to get increasingly confused the longer the reasoning goes on (which is documented by experiments), then the faster the reasoning steps occur the more humans can reason. This might permit all sorts of new conclusions that human beings couldn't make before, and give effects far more than proportionate to the speedup.

7-3.(a) Yes, it's often typical; for instance, many auto repair books speak only in terms of symptoms and fixes. One reason is that people often have trouble thinking up the appropriate intermediate predicates, because they're often abstract and they're not strictly necessary. This presents a problem in building expert systems, because they can't be efficient unless intermediate predicates are defined.

(b) Intermediate predicates are a form of abstraction, and abstraction makes software easier to build, easier to understand, and more reliable. While abstraction does have a cost overhead in design, and also perhaps an overhead in implementation too, good software engineering practice has long emphasized its importance and necessity.

7-4. [Too long to give here]

7-6. [Too long to give here]

7-7. Deletion can be done whenever all ways of using a rule have been exhausted. That occurs when either:

--the exact left side of the rule is now a fact;

--there exists another rule with the exact same left side, whose right side predicate expressions are exactly a subset of those of this rule;

--the rule was created from another rule by matching a predicate expression without binding variables, or only binding variables that did not occur on the left side of the rule;

--the rule contains a "not" whose argument is now a fact;

--the rule right side includes a predicate name that does not occur in a fact or in the left side of any other current rule;

--the rule right side contains a predicate expression with at least one variable, with a predicate that only appears in facts, and every possible binding of the variable(s) has been used to generate a new rule;

--some rule-based systems may have rules in partitions such that if no facts of a particular type are present, then the rules of that partition are useless;

--some rule-based systems may have rules in "levels" such that once all the initial facts have been considered, all the bottom-level rules that refer only to facts can be eliminated. 7-8. [Too long to give here]

7-9. [Too long to give here]

7-10. [Too long to give here]

7-12. [Too long to give here]

7-13. Just change the first term on the right side of the "pursuit" rule to "pick_rule(L,R)", defined analogously to the hybrid version:

pick_rule(L,R) :- rule(L,R), not(better_rule(L,R)). better_rule(L,R) :- rule(L2,R2), prefer(L2,R2,L,R).

7-14. (a) Each "prefer" rule represents conditions for ruling out a rule. So just write another one:

prefer(L,R,L2,R2) :- proved(L2), not(proved(L)). proved(F) :- fact(F). proved(F) :- usedfact(F).

(b) Use the meta-rule:

prefer(L,R,L2,R2) :- not(last_used(L,R)), last_used(L2,R2).

where we modify "pick_rule" as follows:

pick_rule(L,R) :- rule(L,R), not(better_rule(L,R)), checkretract(last_used(L2,R2)), asserta(last_used(L,R)).

where "checkretract" is defined as

checkretract(F) :- call(F), retract(F). checkretract(F) :- not(call(F)).

7-15. To make this simpler, let askif(device_dead) = a, askif(lights_out) = b, askif(smell_smoke) = c, askif(heats) = d, askif(has('knobs or switches')) = e, askif(knobs_do_something) = f, askif(powerful) = g, askif(hear(pop)) = h, askif(cord_frayed) = i, askif(handyperson) = j, askif(familiar_appliance) = k, askif(device_on) = l, askif(has('an on-off switch or control')) = m.

Then after substitution of intermediate predicates, and elimination of redundant rules (two rules for cord breaks), 18 rules remain:

diagnosis('Fuse blown') :- a, b. diagnosis('Fuse blown') :- e, not(f), b. diagnosis('Fuse blown') :- c, not(d), b. diagnosis('Fuse blown') :- c, not(g), b. diagnosis('Fuse blown') :- a, h. diagnosis('Fuse blown') :- e, not(f), h. diagnosis('Fuse blown') :- c, not(d), h. diagnosis('Fuse blown') :- c, not(g), h. diagnosis('Break in cord') :- a, i. diagnosis('Break in cord') :- e, not(f), i. diagnosis('Break in cord') :- c, not(d), i. diagnosis('Break in cord') :- c, not(g), i. diagnosis('Device not turned on') :- a, not(j), m, not(l). diagnosis('Device not turned on') :- a, not(k), m, not(l). diagnosis('Device not turned on') :- e, not(f), not(j), m, not(l). diagnosis('Device not turned on') :- e, not(f), not(k), m, not(l). diagnosis('Device not turned on') :- c, not(d), not(j), m, not(l). diagnosis('Device not turned on') :- c, not(g), not(j), m, not(l). diagnosis('Device not turned on') :- c, not(d), not(k), m, not(l). diagnosis('Device not turned on') :- c, not(g), not(k), m, not(l).

From counting the occurrences of terms on right sides, we have the following statistics (assuming, as was stated, omission of a fact in a rule means it must be false if it occurs true in other rules, and means it must be true if it occurs false in other rules):

(predicate,count-of-occurrences, count-of-negation-occurrences): a,5,15 b,4,16 c,10,10 d,15,5 e,5,15 f,15,5 g,15,5 h,4,16 i,4,16 j,16,4 k,16,4 l,12,8 m,8,12

Predicate c seem be the most even split, so we'll pick it for the first question in the decision lattice. A "yes" answer to question c means that one of these rules applies:

diagnosis('Fuse blown') :- c, not(d), b. diagnosis('Fuse blown') :- c, not(g), b. diagnosis('Fuse blown') :- c, not(d), h. diagnosis('Fuse blown') :- c, not(g), h. diagnosis('Break in cord') :- c, not(d), i. diagnosis('Break in cord') :- c, not(g), i. diagnosis('Device not turned on') :- c, not(d), not(j), m, not(l). diagnosis('Device not turned on') :- c, not(g), not(j), m, not(l). diagnosis('Device not turned on') :- c, not(d), not(k), m, not(l). diagnosis('Device not turned on') :- c, not(g), not(k), m, not(l).

Here predicate expressions m, not(l), not(j), and not(k) are separately sufficient to distinguish the not-turned-on conclusion. Further questions can be asked to validate it. Otherwise, if predicate expression i is true, further questions can be asked to validate a break in the cord. Otherwise, we can try to validate a blown fuse.

Now suppose c is false. Then the applicable rules are:

diagnosis('Fuse blown') :- a, b. diagnosis('Fuse blown') :- e, not(f), b. diagnosis('Fuse blown') :- a, h. diagnosis('Fuse blown') :- e, not(f), h. diagnosis('Break in cord') :- a, i. diagnosis('Break in cord') :- e, not(f), i. diagnosis('Device not turned on') :- a, not(j), m, not(l). diagnosis('Device not turned on') :- a, not(k), m, not(l). diagnosis('Device not turned on') :- e, not(f), not(j), m, not(l). diagnosis('Device not turned on') :- e, not(f), not(k), m, not(l).

Again predicate expressions m, not(l), not(j), and not(k) are sufficient to distinguish the not-turned-on condition, and expression i for the break-in-cord condition.

The final decision lattice can be described this way:

1. Ask if the user smells smoke. If yes, go to step 2; else go to step 3.

2. Ask if the device has an on-off switch or control. If yes, go to step 4; else go to step 5.

3. Ask if the device has an on-off switch or control. If yes, go to step 8; else stop.

4. Verify if the user is either not good at fixing things or is not familiar with the appliance, and if the device is not turned on; if so, that's the problem. Otherwise, stop.

5. Ask if the cord is frayed. If so, go to step 6; else go to step 7.

6. Verify that either the device does not heat or does not take a lot of power; if so, it's a break in the cord. Otherwise, stop.

7. Verify that (1) either the device heats or takes a lot of power, and (2) the lights are out in the house or the user heard a pop sound; if both are true, it's a blown fuse. Otherwise, stop.

8. Verify that (1) either the device seems dead, or the device has knobs and switches and they don't do anything; (2) the user is either not good at fixing things or is not familiar with the appliance; and (3) the

device is not turned on. If all are true, the last is the problem. Otherwise, stop.

9. Ask if the cord is frayed. If so, go to step 10; else go to step 11.

10. Verify that either (1) the device is dead, or (2) the device has knobs or switches and they don't do anything; if so, there's a break in the cord. Otherwise, stop.

11. Verify that (1) either the lights are out or the user heard a pop sound, and (2) either the device is dead or the device has knobs or switches that don't work; if both are true, it's a blown fuse. Otherwise, stop.

7-16. [Too long to give here]

7-17. One way to do it, among many ways, with sample data built in (Chapter 10 will develop a better way to write similar state-changing programs):

```
robot :- repeat, move, goal_reached. goal_reached :- position(30,50). move :- position(X,Y),
stepoptions(DX,DY), X2 is X + DX, Y2 is Y + DY, trymove(X2,Y2), retractblocks(DX,DY), !.
stepoptions(1,0) :- position(X,Y), X < 30, not(verticalblock). stepoptions(0,1) :- not(horizontalblock).
stepoptions(-1,0) :- verticalblock. stepoptions(-1,0) :- not(verticalblock), asserta(verticalblock),
write([verticalblock,asserted]). stepoptions(0,-1) :- horizontalblock. stepoptions(0,-1) :-
not(horizontalblock),asserta(horizontalblock), write([horizontalblock,asserted]). trymove(X,Y) :-
newcraters, not(cratered(X,Y)), not(ravined(X,Y)), withinarea(X,Y), retract(position(X2,Y2)),
asserta(position(X,Y)), time(T), T2 is T + 1, asserta(time(T2)), retract(time(T)), write([x,X,y,Y,t,T2]),nl.
retractblocks(0,1) :- verticalblock, retract(verticalblock), write([verticalblock,retracted]). retractblocks(-
1,0) :- horizontalblock, retract(horizontalblock), write([horizontalblock,retracted]).
retractblocks(DX,DY). cratered(X,Y) :- crater(X2,Y2), withintwo(X,X2,Y,Y2), write('We are on the edge
of a crater.'), nl. withintwo(X,X2,Y,Y2) :- deviation(X,X2,DX), DX < 2, deviation(Y,Y2,DY), DY < 2.
deviation(A,B,D) :- D is A - B, D >= 0. deviation(A,B,D) :- D is B - A, D >= 0. newcraters :- time(45),
position(X,Y), X2 is X - 3, asserta(crater(X2,Y)). newcraters :- time(73), position(X,Y), X2 is X + 4,
asserta(crater(X2,Y)). newcraters. ravined(X,Y) :- Y = 30, X > 9, write('We are on the edge of a ravine.'),
nl. withinarea(X,Y) :- X > -1, X < 51, Y > -1, Y < 51. position(25,0). time(0). crater(20,10). crater(10,25).
crater(20,40).
```

7-18. Causal chain reasoning is only desirable when greater flexibility of reasoning is needed. For instance, when many diseases share common symptoms, and it's hard to give a specific conjunction of symptoms for each. The appliance program is a simplification in which only major symptom-to-cause paths are recorded--that is, summary is done of many causal links. So clearly a better program could reason at a finer level of detail.

8-1. The probability the battery is defective from the second rule is 0.9*0.8 = 0.72. Doing an orcombine with the number from the first rule, we get 1 - 0.5*0.28 = 0.86.

8-2. The first rule gives the probability of "a" as 0.6, and the second as 0.8. We must "orcombine" these two numbers to get a cumulative probability, since they are two pieces of evidence confirming the same thing.

(a) 1 - (1-0.6)(1-0.8) = 1 - 0.08 = 0.92

(b) max(0.6,0.8) = 0.80

(c) min(1,0.6+0.8) = min(1,1.4) = 1.00 8-7. (a) PR (product of the probability and the number of things it can apply to)

(b) Among the R/D rules for a diagnosis, the probabilities of failure are independent too. So the total probability that all R/D rules fail is (1-P) to the R/D power. Hence the probability of that the diagnosis succeeds (it just takes one rule) is 1-(1 - P) to the R/D. That is the inverse of the probability that none of the rules proving a diagnosis will succeed. So the average number of diagnoses that succeed is D(1-(1-P) to the R/D.

8-8. The simplest way is to consider only direct positive evidence for some prediction (that is, no "cross terms").

```
predict(W,P) :- bagof(X,total_predict(W,X),XL), orcombine(XL,P). total_predict(W,P) :-
bagof(X,previous_predict(W,X),XL), orcombine(XL,P). total_predict(W,P) :-
bagof(X,current_predict(W,X),XL), orcombine(XL,P). current_predict(W,0.7) :- current_west_view(W).
current_predict(W,0.5) :- current_east_view(W). current_predict(cloudy,0.9) :- raining(X).
previous_predict(W,0.3) :- weatherman_prediction(W). previous_predict(W,0.1) :- grandma_memory,
grandma_prediction(W). previous_predict(W,0.8) :- secretary_has_radio, not(secretary_out_to_lunch),
radio_prediction(W).
```

Then to use these rules, just query:

```
?- predict(W,P).
```

type semicolons, and the probabilities obtained tell you how likely each kind of sky is. For instance, if you have the facts

```
current_west_view(cloudy). current_east_view(sunny). weatherman_prediction(partly_cloudy).
secretary_has_radio. radio_prediction(sunny).
```

then you get:

```
?- predict(W,P). W=cloudy, P=0.7; W=partly_cloudy, P=0.3; W=sunny, P=0.9;
```

8-10. (a) The problem arises because students are not typical adults, but are exceptions to a usually-true rule. So just as with the traffic lights program we can write an exception rule and modify the old rule accordingly, as for instance:

employed(X,P) :- student(X,P2), P2>0.1, P is 1.05-P2. employed(X,P) :- adult(X,P2), student(X,P3), P3=<0.1, P is P2 * 0.9.

We can also do much the same thing with just one rule:

employed(XP) :- adult(X,P2), student(X,P3), P is P2*(1-P3)*0.9.

but then we have to commit ourselves to a theory of how the two probabilities interact.

(b) The conservative assumption applied to Joe gives 0.9 + 0.95 - 1 = 0.85. Not much less than the independence-assumption value of 0.855.

(c) The problem is that rule-strength probabilities is not quite the same as additional conjuncts in a rule, for purposes of probability combination. So this is different than a rule that says X is employed if X is an adult and X pays property taxes, where each of those two things has an associated probability. A true combination for rule strength the value might be 0, because the evidence of the incoming conclusion probability could be arbitrarily diluted in reaching the second conclusion. (Actually, it's never possible to exactly get 0, but 0 is a lower bound.)

One general way to handle situations like this is to extend evidence-combination to use more than just probabilities as inputs. Probability rules could return not just a probability, but a list of predicates that succeeded in getting the probability. Then a rule like "employed" above can check whether it is particularly sensitive to anything in the list (by checking prestored sensitivity-pair facts), and fail if so; special-case rules for "employed" could then do a different calculation. Another approach is to use Bayes' Rule to estimate probabilities: store certain probabilities of two things happening simultaneously, and reason from those.

8-11. (a) "Happy" and "unhappy" are not true opposites in English--a person could have neither word apply to them, so the two words don't describe possible emotional states. Notice that this is different from the words "done" and "undone" which do represent true opposites. So you sometimes have to be careful analyzing English words.

(b) It is dangerous when building rule-based expert systems with probabilities when the probability of X (e.g. "happy") plus the probability of not(X) (e.g. "unhappy") isn't 1.0. Then there's a three-valued logical situation here: yes, no or neither. So you couldn't get one from the other by subtracting the other from 1. Instead we must reason separately about each of the two rules, writing separate rules about the value of each.

8-12. [Too long to give]

9-1. (a) A state must be a snapshot of a single time, not several times.

(b) Temperature varies continuously--there can be no abrupt changes in temperature. (But a physicist might call this a physical state.)

(c) It's too complicated to describe as just a list of facts. Also, it seems to involve continuous-valued, not discrete, phenomena.

9-3. No. For example, this set of branches:

branch from "a" to "b" branch from "a" to "c" branch from "b" to "d" branch from "b" to "e" branch from "c" to "d" branch from "c" to "e" branch from "d" to "f" branch from "e" to "f"

9-4. (a) A simple way is to abandon the focus-of-attention approach, and put new facts at the end of facts. In our implementation of breadth-first search in the next chapter we'll use this same trick.

(b) A breadth-first approach might be to check all rules whose left sides match query predicates to see if their right sides are all facts: then check all rules whose left sides match the terms of all the previous right sides are all facts and so on. In other words something that sounds suspiciously like the rule-cycle hybrid. The only difference is that rules are being invoked top-down, so an unsuccessful rule need not be tried over and over again. Actually, such an invocation scheme would be a good way to organize hybrid chaining to make it more efficient.

9-5.(a) depth-first, since it would be awkward and silly to constantly return to previous nodes in real time. This is a good example of the difference between computer environments and the real world: it's easy to return to previous nodes on a computer, but hard in the real world. Note of course you shouldn't use plain depth-first search while driving, but you should use heuristics like those of part (i) to avoid getting too far away from the goal.

(b) plan routes from several other places to the goal, places just off the route

9-6.(a) state b (sequence is a, e, s, b). Costs and evaluation function values are irrelevant.

(b) state r (sequence is a, b, o, r). Costs are irrelevant.

9-7. (a) Vertical height on the page can be measured with numbers, but you can also use "above(X,Y)" to represent relationships on the page, to the same effect.

(b) Heuristics are binary: they either must apply or not apply. So to every possible permutation of success or failure of heuristics, assign a number. Store these numbers in a big table, and just look them up as an evaluation function. This evaluation is discrete and not continuous as a better one might be, but it's perfectly acceptable; you may be able to fit a curve to the numbers to get a continuous evaluation function. Note you must rank permutations of success/failure for all the heuristics, not just rank the heuristics, because a branch choice is usually determined by several heuristics working together. It also won't work in general to assign weights to heuristics and sum up the weights for the heuristics that apply to a state, because there are many ways to get the same sum.

(c) The mapping can often be done if you know all the possible states that occur in a problem. Then you can just write very specific heuristics that say if state X occurs then do operator Y, where this recommendation of Y is consistent with the evaluation function. But if you don't know all the possible states or if there are an infinity of them, you can't do this mapping. For instance, you can do this mapping for route planning in a city, because there are a finite number of intersections in a city and they're the only points at which choices can be made; but you can't for route planning through a forest because there are an infinite number of choices you can make about your route.

9-9. (a) branch-and-bound search

(b) best-first search

(c) breadth-first search

9-12.(a) Here are some:

--Prefer the branch that moves terminals only to their proper floor. --Prefer branches that move two terminals instead of one. --Prefer branches that move terminals to adjacent floors (but that may not be a good heuristic, since it may cause you to move terminals twice).

(b) One way is to count the number of terminals on wrong floors and divide by two (two is the maximum number of terminals that can be moved in any one step). But this doesn't take into account that terminals two floors away are going to cost 1.2 times more, so a better evaluation function (that is still a lower bound) would be to sum up over all terminals .5 for each terminal that is one floor away from its correct floor, and .6 for each terminal that is two floors away from its correct floor. (Note that an "evaluation function" is a function definition, not a number.)

(c) Yes, branching factors are similar in both directions. This is because the operators are all reversible (if you take the elevator and some terminals somewhere, you can always take them back). Note you could use A* in both directions with bidirectional search, so A* is not incompatible with bidirectional (and note this would mean fewer node expansions than A* just one-way). It's true there are three goal states (the elevator can be in three possible places), but that just means you shouldn't search as deep going backwards as going forwards, not that bidirectional search won't work.

(d) There are F different floors on which to place a single terminal. There are T terminals to place. So for

T=1, there are F states; for T=2, F*F states; and for F=3, F*F*F states. There are also F places to put the elevator. So in general the number of different states is F multiplied by itself T+1 times, or F to the T+1 power. However, you could argue that all the terminals destined for a single floor are indistinguishable. This would reduce the preceding number since then some of the states considered would be identical, but figuring out exactly how many involves some difficult mathematics, so the above upper bound (or something close) is sufficient for this answer.

(e) Finding a good way to represent the state was very important for this problem. A good way is to code the terminals with digits representing what floor they belong on, and bring together all the terminals on one floor into a list. So states can be represented as [First,Second,Third,Elevator] where the first three terms are lists containing the numbers 1, 2, and 3 an indefinite number of times, and Elevator is the number of the floor the elevator is on. Then the starting state is [[2,3,3],[1],[2],1], which has evaluation 2.5 with the first-mentioned evaluation function, and cost 0 for a total of 2.5. Notice that your state representation must indicate the position of each terminal individually because a state description should contain all the information necessary to uniquely specify a state. So it wouldn't do, say, to only indicate for each floor how many terminals are there, because some of the terminals there might be destined for different floors, and your plan to move them would be different than with other combinations of the same number of terminals on that floor.

From the starting state we can go to one of eight states (using again the first-mentioned evaluation function):

[[3,3],[1,2],[2],2] -- evaluation 2.0, cost 1, total 3.0 [[3,3],[1],[2,2],3] -- evaluation 2.5, cost 1.2, total 3.7
[[2,3],[1,3],[2],2] -- evaluation 2.5, cost 1, total 3.5 [[2,3],[1],[2,3],3] -- evaluation 2.0, cost 1.2, total 3.2
[[2],[1,3,3],[2],2] -- evaluation 2.5, cost 1, total 3.5 [[2],[1],[2,3,3],3] -- evaluation 1.5, cost 1.2, total 2.7
[[3],[1,2,3],[2],2] -- evaluation 2.0, cost 1, total 3.0 [[3],[1],[2,2,3],3] -- evaluation 2.0, cost 1.2, total 3.2

The sixth state is the lowest-cost, so we choose it for expansion. It leads to six new states (remember, there's no point writing down states we found earlier):

[[2,2],[1],[3,3],1] -- evaluation 1.5, cost 2.4, total 3.9 [[2],[1,2],[3,3],2] -- evaluation 1.0, cost 2.2, total 3.2
[[2,3],[1],[2,3],1] -- evaluation 2.0, cost 2.4, total 4.4 [[2],[1,3],[2,3],2] -- evaluation 2.0, cost 2.2, total 4.2
[[2,2,3],[1],[3],1] -- evaluation 2.0, cost 2.4, total 4.4 [[2],[1,2,3],[3],2] -- evaluation 1.5, cost 2.2, total 3.7

All these totals are larger than than the 3.0 figure found for the first and seventh nodes of the first expansion. By heuristic, choose the first of these two. This expands to:

[[1,3,3],[2],[2],1] -- evaluation 1.5, cost 2, total 3.5 [[3,3],[2],[1,2],3] -- evaluation 2.0, cost 2, total 4.0
[[3,3],[1],[2,2],3] -- evaluation 2.5, cost 2, total 4.5 [[1,2,3,3],[],[2],1] -- evaluation 2.0, cost 2, total 4.0
[[3,3],[],[1,2,2],3] -- evaluation 2.5, cost 2, total 4.5

If we choose to use the other evaluation function mentioned in part (b) -- counting 0.5 for each terminal one floor away from its correct floor, and 0.6 for each terminal two floors away from its correct floor -- we get slightly different evaluations. For the first expansion, the states are respectively evaluated to 2.2 (total 3.2), 2.7 (total 3.9), 2.6 (total 3.6), 2.1 (total 3.3), 2.5 (total 3.5), 1.5 (total 2.7), 2.1 (total 3.1), and 2.1 (total 3.3). So the sixth state is again chosen for expansion, and its possibilities evaluate to 1.5 (total 3.9), 1.0 (total 3.2), 2.1 (total 4.5), 2.0 (total 4.2), 2.1 (total 4.5), and 1.5 (total 3.7). Again, all these totals are more than a node in the first expansion, but this time it's the seventh node in the first expansion. So we expand that seventh node, getting these new states:

[[1,3],[2,3],[2],1] -- evaluation 1.6, cost 2, total 3.6 [[3],[2,3],[1,2],3] -- evaluation 2.2, cost 2, total 4.2
[[2,3],[1,3],[2],1] -- evaluation 2.6, cost 2, total 4.6 [[3],[1,3],[2,2],3] -- evaluation 2.6, cost 2, total 4.6
[[3,3],[1,2],[2],1] -- evaluation 2.2, cost 2, total 4.2 [[3],[1,2],[2,3],3] -- evaluation 1.6, cost 2, total 3.6
[[1,2,3],[3],[2],1] -- evaluation 2.1, cost 2, total 4.1 [[3],[3],[1,2,2],3] -- evaluation 2.7, cost 2, total 4.7
[[1,3,3],[2],[2],1] -- evaluation 1.7, cost 2, total 3.7 [[3],[2],[1,2,3],3] -- evaluation 1.7, cost 2, total 3.7
[[3],[1],[2,2,3],3] -- evaluation 2.1, cost 2, total 4.1

9-14. Jigsaw puzzles:

(a) all possible "partially completed puzzles", partitions of a set of puzzle pieces into physically connected groups

(b) trying to attach one side of one piece to one side of another piece

(c) a state where each piece is in its own physically-connected partition

(d) a state where all pieces are in one physically-connected partition

(e) usually. If pieces have distinct colors and textures, you can group pieces by color or texture and solve subpuzzles for each color, then connect the subpuzzles.

(f) no, because two pieces you could compare previously might get connected instead to other pieces, and no longer be available

(g) one solution

(h) depends on the puzzle, but typically 4n*4n, n the number of puzzle pieces

(i) it decreases to zero, at a progressively slower rate, because the number of sites to connect pieces decreases with each piece connection

Symbolic integration:

(a) all conceivable algebraic expressions and their integrals

(b) rules of integration, as are given in tables

(c) some particular expression with an integral

(d) any equivalent expression without any integrals

(e) yes--an integral of a sum (or difference) is the sum (or difference) of the integrals.

(f) no, rules require very specific forms, and use of one rule may make another no longer possible

(g) usually the best (simplest) form is desired

(h) depends on the problem

(i) varies with the problem, but will increase if the size of the expression increases. No obvious tendencies over time.

9-16.(a) (iv), A* search--we have both partial costs (the number of machine instructions of a partial allocation) and a lower-bound evaluation function (the number of variable mentions unassigned).

(b) (ii). The first is an evaluation function, and the third is a control strategy. The fourth is something you do automatically in any kind of allocation search (it's part of the definition of the word "allocation" and hence part of the definition of the operator), and what you'd do anyway can't be a heuristic.

(c) (i), always decreases, since each assignment rules out using that variable mention in future states. And nothing ever increases the choices available at some point.

9-17.(a) All theorems of the given postulate-and-assumption form.

(b) If there are K statements, there are K!/(K-2)!2! = K(K-1)/2 ways of taking a pair. Each step will add one more statement. Assuming we start with N postulates, the branching factor for the step S will have an upper bound of (N+S-1)(N+S-2)/2, or approximately (N+S) squared over 2.

(c) One answer: Try to combine the simplest statements that have symbols in common.

(d) One answer: the number of common symbols between the closest existing statement and the goal statement. Larger value means closer to goal.

(e) Yes, theorem-proving is standard example of monotonicity.

(f) No, no obvious subparts.

(g) Heuristics, because it's very hard to quantify closeness to the goal.

(i) No, because A* requires a lower bound on the number of steps to the goal, not an approximation.

(j) If we had a good evaluation function.

(k) Yes, if branching factor seems reasonable in backwards direction (but it will be if we assume all steps involve combination with an original statement, which is true most of the time).

9-18. (a) Suppose we have N data points with K study variables. Then a state for this problem can be expressed as a list of N data points where the first K study variables are the same as the K study variables in the starting state but where there may be additional study variables reflecting operations performed; plus the same target variable for all states. The set of all such states is the search space. (The size of the search space is infinite, because there are an infinite number of combinations and operations you can perform--you can keep squaring a number forever.) Remember, a state must include everything relevant to describe a "snapshot" of activity in the course of a problem, so you must include all the data points in a single state, and define operators to apply to every data point together (i.e., vector operations).

(b) The operators are just the operations described, as for instance squaring all the values for some study variable to get a new study variable, or taking the corresponding product of two study variables to get a new study variable.

(c) The branching factor for any state is infinite because you could use an infinite number of constants in the first of the three described classes of operators.

(d) The initial state has two successors:

data_point([1,2,1],1). data_point([2,0,4],16). data_point([3,1,9],81).

data_point([1,2,4],1). data_point([2,0,0],16). data_point([3,1,1],81).

And the first of these has three successors:

data_point([1,2,1,1],1). data_point([2,0,4,4],16). data_point([3,1,9,9],81).

data_point([1,2,1,4],1). data_point([2,0,4,0],16). data_point([3,1,9,1],81).

data_point([1,2,1,1],1). data_point([2,0,4,16],16). data_point([3,1,9,81],81).

You may not want to count the first state though--it's pointless. The second of the original successors also has three successors:

data_point([1,2,4,1],1). data_point([2,0,0,4],16). data_point([3,1,1,9],81).

data_point([1,2,4,4],1). data_point([2,0,0,0],16). data_point([3,1,1,1],81).

data_point([1,2,4,16],1). data_point([2,0,0,0],16). data_point([3,1,1,1],81).

And you may not want to count the second state here because it doesn't make much sense, nor the first state here because it duplicates (except for a switch of the third and fourth study variables) the second state of the previous group of three.

(e) Statisticians have a number of such criteria. One simple one is that the "least squares" deviation, the average square over all data points of the difference between the target variable value and its corresponding value for the closest study variable, is less than some small fixed number.

(f) The least squares deviation calculation given in part (e) is a natural evaluation function, since squares are always nonnegative and the value does go to zero when the goal is exactly reached.

(g) It's tricky to find something nonnumeric for choosing branches, because the data itself is numeric. One way is to look at the operators: say prefer any operator that involves two variables to any operator that involves only one variable. Another heuristic is to prefer using new study variables not so far used.

(h) Probably not, because we can't do backwards search easily. For one thing, it's hard to get forwards and backwards search to meet in the middle--the branching factor is so larger. For another thing, we'll have to reason somewhat differently backwards, since we don't know what the goal states will be exactly, just a criterion for checking states to see whether they are goals. We could try to reason forwards from a starting state whose study variables are all but one of the forwards-search study variables plus the original target variable, setting the new target variable to be the remaining one of the original study variables. That would be a kind of backwards searching, and if we found a solution this way we could "reverse" the operators to get a corresponding forwards solution path--it's tricky, but possible. But there would be S ways to do this if there were S original study variables, so it'll be considerably harder searching this way--it won't be worth it if S is large.

(i) Not really, because there's no obvious cost function here--we're concerned about the end result, not the difficulty in getting there.

(j) The square root operator requires that all its values be nonnegative, and the logarithm operator requires that all its values be strictly positive. In addition, the quotient operator requires that none of the values of the second variable be zero. Other than these small restrictions, operators can be used any time for this problem.

(k) No, we have no way to pick nice intermediate goals.

(l) Statisticians probably make use of strong analogies to related problems involving some of the same variables. So they "look up" a few related formulae and test them out sequentially, and usually one of them works pretty well. Occasionally they may have to tinker with the formula as a kind of searching, but this is rare because most phenomena that statisticians attack have been quite thoroughly studied already. You could call this approach a rule-based one. What statisticians do also can often be described as "generate and test" (section 3.12). (They also seem to use means-ends analysis, a technique we'll explain in Chapter 11.)

9-19.(a) You must think ahead to what your opponent would do if you made a particular move. Each move possibility for you has a set of subsequent, and usually different, move possibilities for them. And each of these has a set of replies by you, and so on.

(b) The decisions made on alternate moves (yours and your opponent's) are made according to different criteria: you're trying to win when you're selecting a move, and they're trying to win when they're selecting a move. In terms of the evaluation function, it's like theirs is the negative of yours.

10-2. Here are the necessary definitions:

```
testdepth(L) :- depthsearch([[9,11,9,3,4,5,7,18],[12,16,4,7,2,20,5]],L). goalreached([[],[]]).
successor([LX,LY],[LX2,LY2]) :- member(X,LX), member(X,LY), deleteone(X,LX,LX2),
deleteone(X,LY,LY2). successor([LX,LY],[LX2,LY2]) :- successor2([LX,LY],[LX2,LY2]).
successor([LX,LY],[LX2,LY2]) :- successor2([LY,LX],[LY2,LX2]). successor2([LX,LY],[LX2,LY2]) :-
twomembers([X1,X2],LX), Y is X1 + X2, member(Y,LY), deleteone(X1,LX,LX3),
deleteone(X2,LX3,LX2), deleteone(Y,LY,LY2), length_ok(LX2,LY2). length_ok(L1,L2) :-
length(L1,E1), length(L2,E2), E1 + 2 > E2. twomembers([X,Y],[X|L]) :- member(Y,L).
twomembers([X,Y],[Z|L]) :- twomembers([X,Y],L). deleteone(X,[X|L],L) :- !. deleteone(X,[Y|L],[Y|L2])
:- deleteone(X,L,L2). /* Note the member predicate is defined in depth-first search file */
```

Here's the program in action, using the depth-first search program.

```
?- testdepth(L).

L=[[[],[]],[[18],[16,2]],[[9,3,18],[12,16,2]], [[9,11,9,3,18],[12,16,2,20]],[[9,11,9,3,7,18],
[12,16,7,2,20]],[[9,11,9,3,5,7,18],[12,16,7,2,20,5]], [[9,11,9,3,4,5,7,18],[12,16,4,7,2,20,5]]] ;

L=[[[],[]],[[9,3],[12]],[[9,3,18],[12,16,2]],[[9,11,9,3,18], [12,16,2,20]],[[9,11,9,3,7,18],[12,16,7,2,20]],
[[9,11,9,3,5,7,18],[12,16,7,2,20,5]],[[9,11,9,3,4,5,7,18], [12,16,4,7,2,20,5]]] ;

L=[[[],[]],[[18],[16,2]],[[11,9,18],[16,2,20]], [[9,11,9,3,18],[12,16,2,20]],[[9,11,9,3,7,18],
[12,16,7,2,20]],[[9,11,9,3,5,7,18],[12,16,7,2,20,5]], [[9,11,9,3,4,5,7,18],[12,16,4,7,2,20,5]]] ;
```

```
L=[[[],[]],[[11,9],[20]],[[11,9,18],[16,2,20]], [[9,11,9,3,18],[12,16,2,20]],[[9,11,9,3,7,18],
[12,16,7,2,20]],[[9,11,9,3,5,7,18],[12,16,7,2,20,5]], [[9,11,9,3,4,5,7,18],[12,16,4,7,2,20,5]]]
```

10-3. (a)

```
goalreached(S) :- member(on(c,b,bolt1),S).
```

(b) One helpful one is

```
eval(S,N) :- burial(b,N2), burial(c,N3), right_bolts(N4), right_relationship(N5), N is N2+N3+N4+N5.
right_bolts(0) :- on(b,X,bolt1), on(c,Y,bolt1). right_bolts(1) :- on(b,X,bolt1), right_bolts(1) :-
on(c,Y,bolt1), right_bolts(2). right_relationship(0) :- on(c,b,bolt1). right_relationship(1).
```

(c) The shortest (and hence the breadth-first) solution is to remove all three parts from the first bolt (part "a" can go anywhere but the other two parts must go on the surface, not on "bolt2"), and then put part "b" followed by part "c" on "bolt1".

10-4. (a) This problem suffers from a common weakness of depth-first search problems: it picks a bad initial line of reasoning and gets so lost in pursuing the implications of this bad reasoning that it takes nearly forever to get around to undoing it. Things are further exacerbated by the problem mentioned in part (b).

(b) This program can't recognize that two descriptions of the same state are identical unless the facts describing that state occur in exactly the same order in both. So what we need is a more sophisticated equality check call it "set_equality", that checks that every member of one list is a member of another list and vice versa. We can then replace the "member" definition with a "set_member" that uses this "set_equality" instead. We can define the latter using the "subset" predicate of Chapter 5:

```
set_equality(L1,L2) :- subset(L1,L2), subset(L2,L1). subset([],L). subset([X|L],L2) :-
singlemember(X,L2), subset(L,L2). singlemember(X,[X|L]) :- !. singlemember(X,[Y|L]) :-
singlemember(X,L).
```

then set_member looks like this:

```
set_member(X,[Y|L]) :- set_equality(X,Y). set_member(X,[Y|L]) :- set_member(X,L).
```

and we just change the "member" in the definition of "depthsearch2" to "set_member".

(c) The heuristic implementation given in this chapter refers only to ordering of rules. We only have two rules for this problem, so there's just two choices for ordering them. The real problem not addressed is how to order the different ways of binding arguments to those rules.

(d) One heuristic would be to prefer operators that affect parts mentioned in the goalreached condition. To do this we could add four additional rules, two corresponding to each of the original successor rules, that apply to the more specific cases involving the "goal" parts c and e, and place these rules before their more general counterparts:

successor(S,[on(c,surface,none)|S2]) :- member(on(c,Y,B),S), not(B=none), cleartop(c,S), delete(on(c,Y,B),S,S2). successor(S,[on(e,surface,none)|S2]) :- member(on(e,Y,B),S), not(B=none), cleartop(e,S), delete(on(e,Y,B),S,S2). successor(S,[on(X,surface,none)|S2]) :- member(on(X,Y,B),S), not(B=none), cleartop(X,S), delete(on(X,Y,B),S,S2). successor(S,[on(c,Z,B2)|S2]) :- member(on(c,Y,B),S), cleartop(c,S), member(on(Z,W,B2),S, not(B2=none), cleartop(Z,S), not(c=Z), delete(on(c,Y,B),S,S2). successor(S,[on(e,Z,B2)|S2]) :- member(on(e,Y,B),S), cleartop(e,S), member(on(Z,W,B2),S), not(B2=none), cleartop(Z,S), not(e=Z), delete(on(e,Y,B),S,S2). successor(S,[on(X,Z,B2)|S2]) :- member(on(X,Y,B),S), cleartop(X,S), member(on(Z,W,B2),S), not(B2=none), cleartop(Z,S), not(X=Z), delete(on(X,Y,B),S,S2).

10-6. [too long to give]

10-8. (a) This cuts in half the number of facts that need to be represented with the "piece_cost" predicate name, since you only have to represent the costs in one direction along a piece of a route.

(b) An infinite loop could occur in cases where piece_cost would fail when there are no facts that would match a query on it. But in the searches that use costs--branch-and-bound and A* search--a successor fact must already exist for two intersections before a cost for going between those intersections is found. So it would be easy to check to avoid infinite loops for this case: just make sure that every successor fact has a corresponding cost fact on the same pair of arguments. You could also get infinite loops if you backtrack into the cost function, but notice that our A* search program (the only program given in the chapter having costs) has a cut symbol after the call to "cost" to prevent exactly this.

(c) With the successor facts, there's no such previous checking as was mentioned in part (b). In fact finding all the possible successors of a state is essential to all search strategies, and such a commutative rule would cause you to get stuck on finding successors of the starting state forever. So you'd never get anywhere.

10-9. Here's one way:

robot(A) :- search([none,at(chute),dusty(o1),dusty(o2), fullbasket(o1),vacuumable(o2)],A).

goalreached([O,at(X)]) :- not(X=chute).

eval([],0). eval([X|L],N) :- piece_eval(X,N2), eval(L,N3), N is N2+N3. piece_eval(dusty(X),6).

```
piece_eval(vacuumable(X),10). piece_eval(fullbasket(X),24). piece_eval(holdingbasketfrom(X),1).
piece_eval(P,0).
```

```
cost([],0). cost([[O|S]|L],C) :- piece_cost(O,C2), cost(L,C3), C is C2+C3. piece_cost(none,0).
piece_cost(vacuum,10). piece_cost(dust,6). piece_cost(pickup,3). piece_cost(dispose,5).
piece_cost(putdown,1). piece_cost(go(chute,Y),8). piece_cost(go(X,chute),8). piece_cost(go(X,Y),3).
```

```
successor([O|S],[vacuum,fullbasket(X)|S3]) :- member(at(X),S), not(member(dusty(X),S)),
failing_delete(vacuumable(X),S,S2), delete(fullbasket(X),S2,S3).
successor([O|S],[dust,vacuumable(X)|S3]) :- member(at(X),S), failing_delete(dusty(X),S,S2),
delete(vacuumable(X),S2,S3). successor([O|S],[pickup,holdingbasketfrom(X)|S]) :- member(at(X),S),
not(member(holdingbasketfrom(X),S)), not(member(vacuumable(X),S)), not(member(dusty(X),S)),
member(fullbasket(X),S). successor([O|S],[dispose|S2]) :- member(at(chute),S),
member(holdingbasketfrom(X),S), failing_delete(fullbasket(X),S,S2). successor([O|S],[putdown|S2]) :-
member(at(X),S), not(member(fullbasket(X),S)), failing_delete(holdingbasketfrom(X),S,S2).
successor([O|S],[go(X,Y),at(Y)|S2]) :- failing_delete(at(X),S,S2), member(Y,[chute,o1,o2]), not(X=Y).
```

```
member(X,[X|L]). member(X,[Y|L]) :- member(X,L).
```

```
delete(X,[],[]). delete(X,[X|L],L) :- !. delete(X,[Y|L],[Y|M]) :- delete(X,L,M).
```

```
failing_delete(X,[X|L],L) :- !. failing_delete(X,[Y|L],[Y|M]) :- failing_delete(X,L,M).
```

Here's this program running with the A* program:

```
?- robot(A). 73 incompletely examined states and 255 examined states A=[[putdown,at(o2)],
[go(o1,o2),at(o2),holdingbasketfrom(o2)], [putdown,at(o1),holdingbasketfrom(o2)],
[go(chute,o1),at(o1),holdingbasketfrom(o2), holdingbasketfrom(o1)],
[dispose,at(chute),holdingbasketfrom(o2), holdingbasketfrom(o1)],
[go(o2,chute),at(chute),holdingbasketfrom(o2), fullbasket(o2),holdingbasketfrom(o1)],
[pickup,holdingbasketfrom(o2),fullbasket(o2),at(o2), holdingbasketfrom(o1)],
[vacuum,fullbasket(o2),at(o2),holdingbasketfrom(o1)],
[dust,vacuumable(o2),at(o2),holdingbasketfrom(o1)],
[go(chute,o2),at(o2),holdingbasketfrom(o1),dusty(o2), vacuumable(o2)],
[dispose,at(chute),holdingbasketfrom(o1),dusty(o2), vacuumable(o2)],
[go(o1,chute),at(chute),holdingbasketfrom(o1), fullbasket(o1),dusty(o2),vacuumable(o2)],
[pickup,holdingbasketfrom(o1),fullbasket(o1),at(o1), dusty(o2),vacuumable(o2)],
[vacuum,fullbasket(o1),at(o1),dusty(o2), vacuumable(o2)],
[dust,vacuumable(o1),at(o1),dusty(o2),fullbasket(o1), vacuumable(o2)],
[go(chute,o1),at(o1),dusty(o1),dusty(o2), fullbasket(o1),vacuumable(o2)],
[none,at(chute),dusty(o1),dusty(o2),fullbasket(o1), vacuumable(o2)]] ;
```

```
73 incompletely examined states and 257 examined states A=[[putdown,at(o1)],
[go(o2,o1),at(o1),holdingbasketfrom(o1)], [putdown,at(o2),holdingbasketfrom(o1)],
[go(chute,o2),at(o2),holdingbasketfrom(o2), holdingbasketfrom(o1)],
[dispose,at(chute),holdingbasketfrom(o2), holdingbasketfrom(o1)],
[go(o2,chute),at(chute),holdingbasketfrom(o2), fullbasket(o2),holdingbasketfrom(o1)],
[pickup,holdingbasketfrom(o2),fullbasket(o2),at(o2), holdingbasketfrom(o1)],
[vacuum,fullbasket(o2),at(o2),holdingbasketfrom(o1)],
[dust,vacuumable(o2),at(o2),holdingbasketfrom(o1)],
[go(chute,o2),at(o2),holdingbasketfrom(o1),dusty(o2), vacuumable(o2)],
[dispose,at(chute),holdingbasketfrom(o1),dusty(o2), vacuumable(o2)],
[go(o1,chute),at(chute),holdingbasketfrom(o1), fullbasket(o1),dusty(o2),vacuumable(o2)],
[pickup,holdingbasketfrom(o1),fullbasket(o1),at(o1), dusty(o2),vacuumable(o2)],
[vacuum,fullbasket(o1),at(o1),dusty(o2), vacuumable(o2)],
[dust,vacuumable(o1),at(o1),dusty(o2),fullbasket(o1), vacuumable(o2)],
[go(chute,o1),at(o1),dusty(o1),dusty(o2), fullbasket(o1),vacuumable(o2)],
[none,at(chute),dusty(o1),dusty(o2),fullbasket(o1), vacuumable(o2)]] ;
```

```
71 incompletely examined states and 333 examined states A=[[putdown,at(o1)],
[go(chute,o1),at(o1),holdingbasketfrom(o1)], [dispose,at(chute),holdingbasketfrom(o1)],
[go(o1,chute),at(chute),holdingbasketfrom(o1), fullbasket(o1)],
[pickup,holdingbasketfrom(o1),fullbasket(o1),at(o1)], [vacuum,fullbasket(o1),at(o1)],
[dust,vacuumable(o1),at(o1),fullbasket(o1)], [go(o2,o1),at(o1),dusty(o1),fullbasket(o1)],
[putdown,at(o2),dusty(o1),fullbasket(o1)], [go(chute,o2),at(o2),holdingbasketfrom(o2),dusty(o1),
fullbasket(o1)], [dispose,at(chute),holdingbasketfrom(o2),dusty(o1), fullbasket(o1)],
[go(o2,chute),at(chute),holdingbasketfrom(o2), fullbasket(o2),dusty(o1),fullbasket(o1)],
[pickup,holdingbasketfrom(o2),fullbasket(o2),at(o2), dusty(o1),fullbasket(o1)],
[vacuum,fullbasket(o2),at(o2),dusty(o1), fullbasket(o1)],
[dust,vacuumable(o2),at(o2),dusty(o1),fullbasket(o1)], [go(chute,o2),at(o2),dusty(o1),dusty(o2),
fullbasket(o1),vacuumable(o2)], [none,at(chute),dusty(o1),dusty(o2),fullbasket(o1), vacuumable(o2)]] ;
```

```
71 incompletely examined states and 336 examined states A=[[putdown,at(o2)],
[go(chute,o2),at(o2),holdingbasketfrom(o2)], [dispose,at(chute),holdingbasketfrom(o2)],
[go(o2,chute),at(chute),holdingbasketfrom(o2), fullbasket(o2)],
[pickup,holdingbasketfrom(o2),fullbasket(o2),at(o2)], [vacuum,fullbasket(o2),at(o2)],
[dust,vacuumable(o2),at(o2)], [go(o1,o2),at(o2),dusty(o2),vacuumable(o2)],
[putdown,at(o1),dusty(o2),vacuumable(o2)], [go(chute,o1),at(o1),holdingbasketfrom(o1),dusty(o2),
vacuumable(o2)], [dispose,at(chute),holdingbasketfrom(o1),dusty(o2), vacuumable(o2)],
[go(o1,chute),at(chute),holdingbasketfrom(o1), fullbasket(o1),dusty(o2),vacuumable(o2)],
[pickup,holdingbasketfrom(o1),fullbasket(o1),at(o1), dusty(o2),vacuumable(o2)],
[vacuum,fullbasket(o1),at(o1),dusty(o2), vacuumable(o2)],
[dust,vacuumable(o1),at(o1),dusty(o2),fullbasket(o1), vacuumable(o2)],
[go(chute,o1),at(o1),dusty(o1),dusty(o2), fullbasket(o1),vacuumable(o2)],
[none,at(chute),dusty(o1),dusty(o2),fullbasket(o1), vacuumable(o2)]] ;
```

no

This program run in one dialect took 336K of heap storage, 5K of global stack, and 22K of local stack, and took 154 seconds to reach an answer.

10-11.(a) 1, if the states form a chain

(b) N-1, for a one-level state tree

(c) 1, as before

(d) N-1, as before

10-13. (a) Here B is the branching factor and K is the level number. As explained in Chapter 9. The number of states in such a search lattice up through and including the level K states is ((B to the K+1)-B) / (B-1). If fraction P of these B states are visited, then the new branching factor is B(1-P), leading to a total number of states ((B to the K+1) ((1-P) to the K+1) - B(1-P)) / (B(1-P)-1).

(b) The ratio of the old to new number of states (the two formulae of the last section) is approximately (B(1-P)-1) / ((1-P) to the K+1 - (1-P)B to the -K) (B-1) for either K or B large. This ratio will be greater than or equal to 1, and we want to ensure that it is more than the ratio of costs for the visited-state checking, (E+C)/C = 1+(E/C).

(c) Now the second ratio is 1+(AK/C) for level K. So since the first ratio does not depend on level, there will now be some threshold level K at which visited state checking no longer pays off. So to decide whether to use it, you have to guess in advance how far the nearest goal state is from the startingstate.

10-14. (a) No, the "not" term in the second rule fulfills the function of a "cut" in the first rule, and the second and third rules will never apply to the same situation.

(b) It doesn't have a logical meaning like the rest of Prolog--it has a procedural meaning, i.e. it's a programming trick. Also, it can have effects far beyond its immediate vicinity, since it prevents backtracking in any rule with the same left side as the one we're in.

10-16.

```
f_of_list([],[]). f_of_list([X|L],[FX|FL]) :- call(f(X,FX)), f_of_list(L,FL).
```

10-17.(a)

```
prefer(X1,Y1,E1,X2,Y2,E2) :- member(zzxxy(A),X1), not(member(zzxxy(A),X2)).
```

(b) Use the "length" definition of Chapter 5 and:

prefer(X1,Y1,E1,X2,Y2,E2) :- length(X1,L1), length(X2,L2), L1<L2.

(c) Sum up the number of changes in each path list, and compare:

prefer(X1,Y1,E1,X2,Y2,E2) :- numchanges(Y1,L1), numchanges(Y2,L2), L1<L2. numchanges([S],0) :- !. numchanges([S1,S2|SL],N) :- numchanges2(S1,S2,N2), numchanges2(S2,S1,N3), numchanges([S2|SL],N4), N is N2+N3+N4. numchanges2([],L,0) :- !. numchanges2([S|L1],L2,N) :- not(member(S,L2)), numchanges2(L1,L2,N2), N is N2+1, !. numchanges2([S|L1],L2,N) :- not(member(S,L2)), numchanges2(L1,L2,N).

(d) The last state to find successors of is not one of the six arguments to "prefer" so we must first figure it out. A way that will usually work is to look at the first item on the agenda (which must have been the last successor found), and find the second-to-last state in its path list. This will only be incorrect when the last state chosen had no successors.

prefer(X1,Y1,E1,X2,Y2,E2) :- recent(X2), not(recent(X1)). recent(S) :- first_agenda_item(X,[S2,S3|SL],E), successor(S3,S). first_agenda_item(X,Y,E) :- agenda(X,Y,E), !. 10-18. Breadth-first search is the tool to use, because it always finds minimum-distance paths. So use the problem-independent breadth-first code from the chapter, and define:

inference_distance(X,Y,N) :- setup_goal(Y), breadthsearch(X,Anslist), length(Anslist,N). setup_goal(G) :- retract(goal(X)), !, asserta(goal(G)). setup_goal(G) :- asserta(goal(G)). successor(X,Y) :- a_kind_of(X,Y). goalreached(X) :- goal(X). length([],0). length([X|L],N) :- length(L).

10-19. (a) The state list supplied to the cost function will be in reverse order for the backwards search. So you must reverse this list of states before calling upon the old cost function in order to get the new cost function.

(b) Just take the negative old evaluation function.

(c) You basically want to reverse the arguments to each successor definition. This is easy when the successor definition is a fact or a rule without arithmetic or side effects. If a successor rule has arithmetic you will have to figure out a reverse arithmetic formula to do the calculation in the opposite direction (something not always possible). If there are side effects like assertions or retractions or file loads or cut symbols. The successor rule must pretty well be rewritten because there is usually no obvious way to convert it to the backwards form.

10-20.(a) Change pick_best_move so that it calls upon a two-argument "prefer" predicate, in the manner of section 10.11:

```
pick_best_move(S,NS) :- bagof(X,successor(S,X),XL), member(NS,XL), not(better_pick_than(NS,XL)).
better_pick_than(S,L) :- member(S2,L), not(S=S2), prefer(S2,S).
```

(b) Just the beginning of the program needs to be changed, so "get_move" is called to generate moves for both players:

```
gamesearch(State) :- goalreached(State), print_win(State), !. gamesearch([-1,M|B]) :-
not(get_move(B,1,S,NB)), write('Player X cannot move so Player O gets another turn.'), nl,
!,gamesearch([1,M|B]). gamesearch([P,M|B]) :- print_board(B), pick_best_move([P,M|B],Newstate),
write_move(Newstate), !, gamesearch(Newstate). gamesearch([1,M|B]) :- not(get_move(B,-1,S,NB)),
not(get_move(B,1,S,NB)), print_win([1,M|B]), !. gamesearch([1,M|B]) :- not(get_move(B,-1,S,NB)),
write('Player O cannot move so Player X gets another turn.'), nl, !, gamesearch([-1,M|B]). gamesearch(S)
:- print_win(S), !. write_move([P,[X,Y]|B]) :- Move is (10*X)+Y, write('Player '), decode(P,C), write(C),
write(' makes move '), write(Move), nl, !.
```

(c) Redefine pick_best_move for one player:

```
pick_best_move([-1|S],NS) :- asserta(ebest(none,1000000)), successor([-1|S],S2), one_ply_eval(S2,E2),
write_eval(S2,E2), ebest(S1,E1), E2<E1, retract(ebest(S1,E1)), asserta(ebest(S2,E2)), fail.
pick_best_move([-1|S],NS) :- ebest(NS,E), retract(ebest(NS,E)). one_ply_eval(S,E) :-
asserta(e2best(1000000,0)), successor(S,S2), eval(S2,E2), e2best(E1,N), min2(E1,E2,E3),
retract(e2best(E1,N)), N2 is N+1, asserta(e2best(E3,N2)), fail. one_ply_eval(S,E) :- e2best(NE,N), E is
(60.0/(N+1.0))-NE, retract(e2best(NE,N)), !.
```

(d) There are many ways to do this.

11-2. You might decompose a problem into five parts when two equally important operators seem suggested for a problem. Two operators that you must do in a particular order. Then the five subproblems are: l) satisfy the conditions of the first 2) apply the first operator 3) satisfy the preconditions of the second operator from there 4) apply the second operator 5) go from this state to the goal state. An example would be the state where both the batteries and the light are defective and you have to replace both, and the replace_batteries and replace_light are equally important. To implement this in Prolog you just create another table, "doublerecommended" that gives conditions for these operator pairs to apply. Then you write a third means_ends rule that comes between the two previous rules that checks the "doublerecommended" rules before trying the regular "recommended" rules, recursing as necessary. This way if no "doublerecommended" rules apply to a situation, the "recommended" rules will be used instead.

11-3. (a) The operators are something like:

If in the current state Sir Reginald is alive and in the goal state he is dead then the recommended operator is to stab him with a knife. If you need to get a knife, then the pick up knife operator is recommended. If you need to put down the knife and hide it, then the hide knife operator is recommended. If your goal involves being in a different room than you're at now, then the go operator is recommended. If you need to go to some other room and that door is locked then the knock operator is recommended.

And the informal preconditions and postconditions are:

preconditions of stabbing with a knife: the person you want to stab is alive and you are in the room with them. Postconditions of stabbing with a knife: the person you want to stab is dead. Preconditions of picking up a knife: you do not have a knife, and you are in the room with a knife. Postconditions of picking up a knife: you have a knife. Preconditions of hiding the knife: you have the knife. Postconditions of hiding the knife: you do not have the knife, and the knife is hidden. Preconditions of going from one room to another: that you are at the first room, and that there is no locked door in between. Postconditions of going from one room to another: you are at the second room. Preconditions of knocking on the door: the door is locked. Postconditions of knocking on the door: the door is unlocked. Preconditions of waiting: there are people in the corridor. Postconditions of waiting: there is no one in the corridor.

(b) The final operator sequence is to wait, pick up the knife, knock on his door, go in when he opens it, stab him, then return to your room and hide the knife.

(c) You could figure out a set of plans instead of just one plan, one for each eventuality. Then in actually executing the plan, you follow one along until there is discrepancy between what facts are true and what facts the plan assumes. Then you transfer to another plan that does not have this discrepancy.

11-5. (a) The answer found is to disassemble the case, turn over the case, replace the batteries, disassemble the top, replace the light, assemble the top, and assemble the case.

(b) Reassembling the case is given a higher priority than replacing the light, so after replacing the batteries the case is reassembled even though it needs to be disassembled to replace the light. So two actions are inserted unnecessarily in the operator lists.

(c) Now fixing the light is given the highest priority and fixing the batteries is given lowest priority, lower even than assembling the case. So fixing the light is done first and two unnecessary actions are inserted in the operator sequence similar to the ones in part (b). The operator sequence is to disassemble the case, disassemble the top, replace the light, reassemble the top, assemble the case, disassemble the case, turn over the case, replace the batteries, and assemble the case.

11-8. (a) A breadth-first approach would be to find the preconditions for a given operator then figure out the operator most relevant to finding those preconditions. But instead of now figuring out the preconditions for this new operator we instead turn our attention to the postconditions of the original operator. While we can't figure out exactly what this state will be (because deeper level of recursions that we haven't looked at yet may add some "extra side effect" facts), we can be sure that everything mentioned in postconditions for the original operator must hold in the state resulting from it. So on that basis we can recursively figure out a recommended operator to get to the goal from there, which implies certain postconditions and preconditions of its own. Now we go back to the precondition recursion of the original operator and figure out how to satisfy its preconditions, then how to satisfy its postconditions. So we gradually build up a list of operators, one level at a time. But as we do we may have to redo some later sections of the operator list as assumptions that they made about their states turn out to be violated by earlier states. So you may have to jump around a bit.

(b) As you can see, this approach is much more awkward than depth-first search and involves an added complication in the necessity to redo work that was done previously on occasion. The control structure of depth-first search is better suited to the way computers work, and while it does have the disadvantage of possibly getting into infinite loops if the rules aren't written properly, this seems unlikely. Depth-first search may also require considerable backtracking, but this application doesn't seem like one where this would happen. Mean-ends analysis is a more "intelligent" kind of search, one that is deliberately set up in an intelligent way. So that the best choice is made first and as often as possible, and backtracking is unnecessary most of the time.

11-9. Mean-ends analysis is search with recursive decomposition based on prerequisites and postconditions of "key" actions. A table is used to define these key actions for all start-goal pairs in the problem domain. So whatever application you found to answer this problem, it must involve: (1) recursive decomposition, not just one level of decomposition; (2) prerequisites and postconditions used to determine the decomposition and each level. And also it must involve (3) operating systems and (4) user-friendliness as stated in the problem.

You can use means-ends analysis to let the user to specify high-level goals (like "Run me off a copy of file p375, wherever it is") without requiring him to worry about the steps needed or the many prerequisites. For this example command, for instance, you could first check to see if the file was in any directory on the current computer. If not, you log in on the next most likely computer (which requires that the computer be up), giving password, and check to see if it's there, and so on until you find it on some computer that is up. Then you log out of the remote computer, and enter the network file transfer program to get the file to the local computer. This may involve figuring out file formats for the remote file and the computer you're on, which may involve looking up information in online directories or manuals. Also, you may not find what you're looking for in the first place in the manual that you look, and you may have to follow up on cross-references. Now you want to print, so you first check to see if the text of the file has formatting commands, in which case you send the file to an appropriate formatter. In any event, you then check to find a printer that is working, and if so, send the file, now in printable form, to that printer. What's happening is that certain goals involve certain subgoals (not all of which you know in advance), which in turn involve other subgoals (which you also may not know in advance), so this is a good situation for recursive decomposition as you proceed.

You can also use means-ends analysis to navigate over complicated computer networks, much in the manner of the route-finding example in Chapter 9. To do this, you would have to consider certain computers as high-level goals that you were trying to reach, and then plan how to reach them, and plan what to do after you reach them. An important example of such a high-level goal computer is a gateway to another network; you could plan to get to that computer, plan to log on and get through to the other network, then plan to get to where you're going on the other network. But if there aren't any such "bottleneck" computers you should probably just do a depth-first or breadth-first search.

11-10. Both involve finding a string of words that solve some problem, in the first case an operator sequence to accomplish some goal in the second case a sentence in some language. But for means-ends analysis, the objective is to discover this string, whereas in parsing the string is already known and you want to find the route to it. Both essentially involve substitution of strings of words for other words, though means-ends analysis is not so obvious about it. Both usually imply depth-first control structure because it's easy to use. But means-ends analysis makes its decision about what to do next based on its analysis of preconditions and postconditions, whereas top-down parsing must rely on heuristics for efficiency because there are too many alternative word sequences for one to accurately speak of preconditions and postconditions.

11-11. [too complicated to give]

12-1.(a) Some possibilities: purpose (filled in with "education"), scheduled (filled in with "yes"), sponsoring-institution (filled in with "university")

(b) Superconcept possibilities: class meeting at any university, meeting of any sort at this school. Subconcept possibilities: class meeting for CS3310, class meeting today.

(c) script inherits from class meeting at any university, and latitude-longitude position inherits from class meeting at this school; location, professor, and course number do not inherit from either.

(d) the classroom-lecture script

12-2. People could carry little credit cards containing all answers to questions they are commonly asked, like name, social security number, address, birth date, employer, etc. Then inheritance could have the physical meaning of "look it up on the card and print it on the form". There could be different levels of inheritance: for instance, in filling out a government form, an employee could provide three credit cards: one themselves, one for their department, and one for their organization, all providing different default values.

12-6. The intension includes definitional information like that it's a form of business communication, it's written in paper with words, and it's usually a formal document. The extension includes the information that memos are common in big organizations and that memos are increasingly being replaced by computer message systems.

12-7. See the next page.

12-8. (a) A value in some slot. Usually the slot is definitional, since adjectives usually help specify something.

(b) The name of a slot. For instance, "of" often signifies a "part_of" relationship; "for" signals a purpose property; "to" signals a destination property or relationship; and "at" signals a directional relationship either in space or in time.

12-10. Note it's important to give unique names to frames (frames are like global variables). It's also important not to use the general English words for instances of a concept; for instance, the frame describing the particular letter should be called something like "letter1" and not just "letter". Also note the experiences reported in the letter occurred in a specific order, and that order must be shown somehow. One solution:

frame-name: letter sender: addressees: subject: reason:

frame-name: letter1 a_kind_of: letter sender: Irate-Programmer addressees: [Fly-by-Nite-Software] subject: refund reason: false-claims1

frame-name: false-claims1 a_kind_of: contract-violation plaintiff: Irate-Programmer defendant: Fly-by-Nite-software summary-report: letter1 violations: [bug1,bug2,bug3]

frame-name: bug product: seller: observer: description:

frame-name: bug1 a_kind_of: bug a_kind_of: numerical-error product: 'Amazing Artificial Intelligence Software' seller: Fly-by-Nite-software observer: Irate-Programmer before: bug2 after: none description: 'attempt to divide by zero'

frame-name: bug2 a_kind_of: bug a_kind_of: lineprinter-output product: Amazing Artificial Intelligence Software seller: Fly-by-Nite-software observer: Irate-Programmer before: bug3 after: bug1 description: '100,000 linefeeds on lineprinter'

frame-name: bug3 a_kind_of: bug a_kind_of: file-system-side-effect product: Amazing Artificial Intelligence Software seller: Fly-by-Nite-software observer: Irate-Programmer before: none after: bug2 description: 'all user files destroyed'

12-11. (a) Intersect the lists inherited. (An empty intersection would mean a contradictory database, so we don't have to worry about it.)

(b) Intersect the ranges inherited, analogous to (a).

12-12. (a) This suggests a priority ranking on the three dimensions, since the user dimension is assumed to have precedence on this issue with most operating systems. Under some circumstances, we might want to leave all the slots empty above our frame in the user hierarchy, in which case the time hierarchy might have the next priority since users are creatures of habit. And if these frames are blank in this slot too, we can defer to the program and facility hierarchy.

(b) Such a warning is very serious business, so any reason at all to give this warning should be heeded. In other words, use the "or" of the Boolean values inherited along the three dimensions.

(c) This parameter is just a guide in allocating resources for the operating system, so just some reasonable estimate is sufficient. That suggests taking the average of the three fractions inherited.

12-14. Rules can be grouped into modules, and each appliance can be characterized by the set of modules that it requires, defining a frame. Thus generalizations over similar appliances (e.g., "drill" and "kitchen appliance") can be characterized by some common set of module frames required by each appliance in the class. Each such frame could be an instance of a multiple-instance slot with the name "rule-module", similar to "part_of". Such module frames will need to also include specifications of when to invoke other frames. They can also include slots for core required, expected execution time, predicate names referred to, a list of facts assumed by all rules in the frame, and an indication of whether backward, forward, or hybrid chaining is best for their rules.

12-15. [many answers possible]

12-16. The general idea is to characterize the two mappings and then compose them (that is, use the results of one mapping as input to the other). The mappings you need to characterize are from A to B and from A to C. To characterize a mapping you must analyze what is different between the two frames in slots and slot values. For this problem, we are especially interested in what new slots are present in the second and are not in the first, and what slot values are filled in in the second frame that are not filled in in the first frame. So we make up a list of such differences from frame A to frame B, a list of differences from frame A between frame C and combine them, eliminating duplicates and finding compromise values for any contradictions. Artificial-intelligence researchers have done some interesting work on this problem.

13-2. Rearrangement of the original query will give a much better execution time than dependency-based backtracking could. To rearrange, first consider which predicates can bind variables. Predicate b cannot bind Y and predicate d cannot bind either X or Z. But the other predicates can be freely moved left in the query. Clearly best among these is predicate e which can match only a single value. It makes predicate a unnecessary, so we can just put e first and c second. So the only remaining question is the order of the final two predicates, b and d. Rule 1 isn't really relevant because we don't know probabilities of success, but using Rule 4 we should put d before b. So the value of Z can be tested right away after the binding of Z. So the best rearrangement of the query is

```
?- e(X), c(Z), d(X,Z), b(X,Y).
```

13-3. The dependencies among predicates in this query form a cycle. There's no way a cycle can be turned into a sequence without eliminating at least one of the dependencies in that original cycle.

13-5. Usually the artist (whether verbal, visual, or whatever) has a general idea which they try to implement in an aesthetic manner. This means making choices about what to do in parts of the whole based on the relationship based on those parts to other parts--in other words constraints. As choices are made further limitations are imposed on later choices so relaxation methods might help.

13-7.(a) examples: sky, cloud, grass, vehicle, road, water, water-vehicle, pier, path, building

(b) Examples: none of these labels can be the same for adjacent regions. Sky and cloud must be above grass, road, water, path, and building. Vehicle must be in or touching road or path, and water-vehicle must be in or touching water. Vehicle cannot be in sky, cloud, or grass; water-vehicle cannot be in sky, cloud, grass, or building. Pier must be touching water, and must also be touching either grass, path, or road. Water cannot be in sky, cloud, vehicle, road, water-vehicle, pier, path, or building. Grass cannot be in sky, cloud, vehicle, road, water, pier, path, or building.

(c) [answer depends on the constraints chosen in (b)]

13-8. The variables are the letters of the acronym; call them N, C, SA, and SB. C1 and C2 are the single-variable constraints, so we apply them first, to get the possibilities: N:[national,naval,new], C:[computer,crazy], SA:[science], SB:[science,school,staff]. Starting with N, "new" is impossible by C6. For C, "crazy" is impossible. For SB, "science" is impossible by C3, and "staff" is impossible by C5. That's all we can do. Final possibilities for the acronym: "national computer science school" or "naval computer science school".

13-9. One way (some steps can be interchanged in the following):

(1) The three small touching objects must be ships by C3.

(2) The region surrounding them must be water by C2.

(3) The "border" region surrounding the entire picture cannot be water by C6, it can't be a cloud by C9, so it's land or marsh.

(4) The small region below the ships is land, marsh, or cloud by C6. But it can't be marsh by C5, so it's land or cloud.

(5) The middle region is land, marsh, or cloud by C6. But it can only be land or cloud by C4 applied to the upper-right T vertex.

(6) The big upper left region is land, marsh, or cloud by C6. But it can only be land or cloud by C4.

(7) The middle region cannot be land any more because marsh has been ruled out for the upper-left

region, using C4 part (b). So it must be a cloud.

(8) That means that the semi-small region upper middle of the picture must be cloud by C4 part (b).

(9) The medium-size region left and below the middle region is land, marsh, or cloud by C6. But it can't be marsh by C4 (since then it would violate both parts of C4), so it's land or cloud. But it can't be land, since then either the border region or the big upper left region would need to be water to satisfy C4 part (b) on the left side, and both those labels are impossible. So it must be cloud.

(10) Hence the small region inside the last-considered region is an airplane.

(11) Hence the lower left region must be a cloud too by C4 part (b) applied at the left of the two T vertices.

(12) The small region in the upper left must be an airplane by C2.

13-10. (a) 8 multiplied ten times, or 8 to the 10th power = 2 to the 30th power, or about one billion.

(b) Each propagation cuts the size of the search space (number of possible combinations) in half, so 30 steps are needed on the average.

13-11. It's like the "dual" of the original picture graph, as they say in graph theory. Regions become vertices, vertices become regions, and edges become edges (though with a different meaning).

13-12. The objective is to find some set of weak constraints that when assumed give a unique solution to the problem. For this search problem, the states are subsets of the set of all weak constraints. Goal states are the states whose constraints, plus the original "strong" constraints, give one and only one interpretation to the problem. An evaluation function is the number of solutions possible with the full set of constraints. A cost function in the total probability that a set of weak constraints hold simultaneously. Computing it may be a little tricky, though; it's a form of "and"-combination, so we can use the conservative independence or liberal assumption from Chapter 8. But a more general approach would be to divide the weak constraints into classes where each class is relatively independent in probability from every other class, then use the conservative assumption to combine probabilities within each class, and the independence-assumption combination to combine those. Search with both an evaluation and cost function suggests the A* search strategy. The evaluation function is not a lower bound on the cost function, so we can't be sure the first solution we find is the optimal solution, but that's all right because we are more concerned about getting a reasonable solution than an optimal solution.

13-15. You can reuse much of the code in the cryptarithmetic example. The last eight lines of that file can be copied directly, the lines concerning satisfiability of the uniqueness condition. The other "satisfiable" rule can look very similar to that for "sum":

```
satisfiable(product,L,[D1,D2,Carryin,Digit,Carryout]) :-
some_bindings(L,[D1,D2,Carryin,Digit,Carryout]), P is (D1*D2)+Carryin, P2 is Digit + (Carryout*10),
P=P2. satisfiable(unique,L,VL) :- unique_values(L,VL,[]).
```

There are eight distinct digits in the problem so let's call them a, c, d, e, f, g, and h. We'll call the carries c1, c2, and so on. The the choices are:

```
choices(D,[0,1,2,3,4,5,6,7,8,9]) :- member(D,[c,d,e,f,g]). choices(D,[1,2,3,4,5,6,7,8,9]) :-
member(D,[a,h]). choices(C,[0,1,2,3,4,5,6,7,8]) :- member(C,[c1,c2,c3,c4,c5,c6,c7]).
```

And the constraints can be written:

```
constraint(product,[h,h,0,a,c1]). constraint(product,[g,h,c1,a,c2]). constraint(product,[f,h,c2,a,c3]).
constraint(product,[e,h,c3,a,c4]). constraint(product,[d,h,c4,a,c5]). constraint(product,[c,h,c5,a,c6]).
constraint(product,[2,h,c6,a,c7]). constraint(product,[a,h,c7,a,a]). constraint(unique,[a,c,d,e,f,g,h]).
```

The solution found is 12345679 * 9 = 111111111.

13-16. [too hard to give]

13-17. [too hard to give]

14-3.(a)

```
(1) on(a,table). (2) on(b,a). (3) on(c,a). (4) on(d,c). (5) color(a,blue). (6) color(b,blue). (7) color(c,red). (8)
color(d,green). (9) above(X,Y); not(on(X,Y)). (10) above(X,Y); not(on(X,Z)); not(above(Z,Y)).
```

(b) The goal is "above(d,Y), color(Y,blue)"; we use proof by contradiction.

(11) Assume the negation of the goal:

```
not(above(d,Y)); not(color(Y,blue)).
```

(12) Resolve 11 with 10, taking X=d and Y=Y, giving

```
not(on(d,Z)); not(above(Z,Y)); not(color(Y,blue).
```

(13) Resolve 12 with 9, taking X=Z and Y=Y, giving

```
not(on(Z,Y)); not(on(d,Z)); not(color(Y,blue)).
```

(14) Resolve 13 with 4, taking c=Z, giving

not(on(c,Y)); not(color(Y,blue)).

(15) Resolve 14 with 5, taking Y=a, giving

not(on(c,a)).

(16) Resolve 15 with 3, giving the null clause.

14-4. No, resolution can only cross out one thing at a time. To see this, suppose we match just the "a" and "not(a)". Then we get as a resolvent "b; not(b)." which is always true (a "tautology"), the exact opposite of the null clause. A tautology is useless because it always is true; it doesn't need to be derived.

14-6. (a)

a; not(b); not(c). a; b; not(d).

(b) a :- c, d.

(c) Given two rules with the same left side, and some term that occurs in unnegated form in one rule and negated form in the other, you can write a new rule with that same left side, whose right side consists of all the terms on both right sides, eliminating duplicates. If furthermore some P and not(P) occur on the new right side, then the left side is a fact.

Before making this new rule a part of the database, we should check that it is not implied by any other single rule. Generally speaking, it's not a good idea to delete the old rules when this new rule is added to the database because it won't often be better than the old rules. (It's likely it has as many or more terms than the old rules.)

14-7. (a) weekend(T); holiday(T); not(empty(spanagel421,T)).

(b) holiday(T) :- not(weekend(T)), empty(spanagel421,T).

(c) (holiday(T),weekend(T),empty(spanagel421,T),full expression): (true,true,true,true) (true,true,false,true) (true,false,true,true) (true,false,false,true) (false,true,true,true) (false,true,false,true) (false,false,true,false) (false,false,false,true)

(d) The original weekend(T) term was unnegated, so the negated form in the new rule must be a "real" or "true" not--something impossible to ensure in standard Prolog. In other words, the new rule will succeed in more situations than the first rule.

14-8. It requires a more sophisticated kind of reasoning than Prolog is capable of without considerable additional support. We could prove that p(7) is true when p(6) is true using Prolog, but we can't prove that p(N+1) is true for some arbitrary N when p(N) is true, because adding the fact "p(N)" to the database would mean N+1 could match N too (query variables are local to a query, and database variables are local to the database rule or fact). The axiom also requires reasoning about a hypothetical situation, something Prolog can't do directly (though creating "possible worlds" in search processes can come close to this). These are "second-order-logic" and "existential quantification" issues, but they're in a different form from that discussed in Section 14.1. A number of resolution-reasoning systems do allow you to add similar axioms to your clauses and reason about them.

14-10. (a) Change the first line of "resolution" to:

```
resolution(C1,C2,Cnew) :- twoclauses(C1,C2),
```

And use a "bagof" to collect all clauses, and then the "append":

```
twoclauses(C1,C2) :- bagof(C,clause(C),CL), append(L1,[C1|L2],CL), append(L3,[C2|L4],L2).
append([],L,L). append([X|L],L2,[X|L3]) :- append(L,L2,L3).
```

(b) Add "fix_subsets(Cnew)" just before the "asserta(clause(Cnew))" in "resolution", defined as:

```
fix_subsets(C) :- not(fix_subsets2(C)). fix_subsets2(C) :- clause(C2), subset(C,C2), retract(clause(C)), fail.
```

15-2. Add an extra argument to all Prolog rules to hold the trace list, and assemble this list using the "append" predicate (section 5.6) as rules succeed. For instance, if an original rule named "r5" was

```
a(X) :- b(X,Y), c(X).
```

then change it to

```
a(X,T) :- b(X,Y,T1), c(X,T2), append([r5|T1],T2,T).
```

And add another argument "fact" to all facts.

15-4. Pick an arbitrary example case for each conclusion you want the rule-based system to make. Then for each case, write a rule whose left side is desired conclusion and whose right side is the conjunction of all facts true for that case. For instance, if diagnosis "a" should be made when facts "b", "c", and "e" are true and "d" is false, write a rule

```
a :- b, c, not(d), e.
```

15-5. (a) R1 could have the same predicate name on its left side as R2, but R1 could have filled in (that is assigned to constants) one or more arguments that are variables in R2.

(b) First we need a definition: a predicate expression P covers a predicate expression Q if either a Prolog query on P can match a database fact Q, or if Q and P can be written respectively as "not(R)" and "not(S)" and S covers R. We can then say that rule R1 is redundant with respect to rule R2 if (1) every predicate expression on the right side of R2 covers an expression on the right side of R1, and (2) the left side of R2 covers the left side of R1. (The right side of R2 may include additional terms not on the right side of R1.)

15-7. [too complicated to give]

M-2. (a) Search can always be thought of as a problem of traversing a lattice.

(b) Frames form lattices based on "a_kind_of" and "part_of" links.

(c) Decision lattices and and-or-not lattices are important compilation tricks. Also, the partitions created when partitioning a rule-based system could form a lattice.

(d) Resolution is a kind of search, and so can be viewed as lattice traversal where each node is a set true clauses at some point. But also a lattice can model the dependencies between clauses, where each node is a single clause: if a clause X is proved by resolving clauses Y and Z, then you can draw arrows from Y to X and from Z to X.

M-6.(a) Yes, if A causes B, and B causes C, then A causes C (though C may have other causes too).

(b) In two cases. But if A causes B, and C is part of B, then A causes C. And if A causes B, and B is part of C, you could argue that A is one of the causes of C, and consistent with our definition of "A causes C". But if A causes B, and A is part of C or C is a part of A, it doesn't make sense that C could cause B.

(c) One thing is that you can't reason backwards with rules; they're a "one-way" (unidirectional) way of reasoning, whereas often it's useful to reason from effects to causes as well as from causes to effects (as in expert systems for diagnosis of problems). So writing causes as two-argument facts

causation(<cause>,<effect>).

would be more flexible. Also, the Prolog ":-" symbol just reflects one aspect of causation, and it doesn't represent well the notion that events have times, and the time of the effect cannot be before the time of the cause.

M-7.(a) "But" is like an "and" or two assertions (that is, two separate assertions) but where the second assertion violates the usual consequences of the first assertion. Typically, the statement "x but y" means that x and y are facts, yet there's a rule with probabilities (or a rule could be written) that says something like "yo(P) :- x.", where "yo" means the opposite of "y", or "yo" implies the opposite of "y" by a chain of forward-chaining inferences.

(b) "But" would be useful in summarizing facts, when certain facts don't make an analogy that usually holds. For instance, the facts

a(1,4). a(1,3) a(1,2). a(7,4). a(7,3).

could be summarized as "Predicate a holds for 1 and 7 as first argument, with 2, 3, and 4 as second argument, but not for (7,2)." "But" would also be useful in describing exceptions to general rules, for instance, "the 'flying-ability' predicate usually inherits with value "yes" for birds, but not for penguins, when it is overridden."

(c) No, because it's too much a natural-language ("discourse") signal used to facilitate communication, and it doesn't have much logical meaning beyond an "and".

M-8(a). Sounds like a classic application for a rule-based system with forwards chaining. Probabilities are essential since evidence is weak. Frames for different kinds of objects and their behavior might help organize facts. Blackboards could help if a lot of hypotheses are generated, as well as focus-of-attention conflict resolution. Generate-and-test could be used to explain sensor readings based on observations, but it makes more sense to use it to write rules than to execute them, which doesn't really count. Inheritance might apply if you have hierarchies of object classes. The other techniques listed aren't very helpful for this problem.

(b) Sounds like a classic search problem. It's a search that tries to optimize something, but apparently no path costs are involved, so best-first search is the only applicable search strategy from the choices given. You could use generate-and-test to guess good solutions and verify them, desirable if you have a good generator provided by an expert in the domain of the problem. If you have a lot of constraints (like protecting certain positions sufficiently), relaxation would be helpful. Frames for your resources (personnel and weapons) could summarize their properties, and you could have hierarchies of them. Inheritance and defaults might be helpful on these hierarchies. The A* algorithm could be used if you took into account costs of steps necessary to achieve a configuration. A rule-based system with either forwards or backwards chaining might suggest good approaches, but wouldn't guarantee optimality of a configuration.

M-9. [too complicated to give]

.PA

ANSWERS TO THE ADDITIONAL PROBLEMS IN THIS MANUAL

I-1. (a) "yes", since X, Y, and Z can be bound to 2

(b) "Q=4". For this X=4, Y=2, and Z=2.

(c) Three times, once for every fact that can match the first predicate expression in the rule, b(X,Y). This makes sense because the interpreter can only answer "no" when it's tried every "b" fact.

I-2. (a) The maximum number of answers occurs when there are the maximum number of ways to link the two predicate expressions. This occurs when there are the minimum number of B values. There can't be only one B value, since then there could only be one fact that could satisfy the query, the fact with both arguments identical. So there must be two possible B values; call them "a" and "b".

For the maximum number of query answers, we shouldn't have anything besides a and b as a first argument in a fact, for otherwise those facts couldn't match any queries. So the first argument to every fact must be either a or b.

This gives as the solution the facts involving the only four possible pairs of the two values:

link(a,a). link(a,b). link(b,a). link(b,b).

For this there are eight answers:

A=a, B=a, C=a; A=a, B=a, C=b; A=a, B=b, C=a; A=a, B=b, C=b; A=b, B=a, C=a; A=b, B=a, C=b; A=b, B=b, C=a; A=b, B=b, C=b;

(b) If the only answer to the query is "no", four matchings for the first predicate expression will be tried. That means three backtracks to get the four answers, plus one more backtrack to fail the first predicate expression and fall off the left end of the query.

I-3. (a;not(c)),b,c = (a,b,c);(not(c),b,c) = a,b using the distributive law and then recognizing an "or" of E with a contradiction must be E.

I-4. (a) R=north, S=hall, T=poindexter

(b) 2 times, because it eventually uses the third fact to match the first predicate expression.

I-5.(a) yes, the extra extra just has the potential to rule out previous answers, and can't add new answers

(b) always more time, since all the same work as before is being done before the new expression is reached

(c) yes, for the same reason; the 26-7=19 new failures are all due to the new expression

(d) now sometimes more and sometimes less, because the new expression could decrease time by quickly rule out cases that took a lot of work to rule out previously

I-6.(a) The fourth rule.

(b) Yes: Q=usnavy, T=enterprise, P=usgovernment.

I-7. (a) Yes, in the sense that if A is necessary for B, and B is necessary for C, then A is necessary for C:

```
necessary(A,C) :- necessary(A,B), necessary(B,C).
```

It doesn't make sense to call this either "upwards" or "downwards", since events can't be more general than other events. Maybe call this "sideways" transitivity.

(b) Yes, if the "before9" property holds for event B, and A is necessary for B to have occurred, then the before9 property holds for event A:

```
before9(X) :- necessary(X,Y), before9(Y).
```

But it doesn't work reasoning in the other direction. Note that even though we wrote the "before9(X)" predicate expression with one argument, you can think of it as equivalent to a predicate "before(X,9amfeb101986)" in the more conventional property-predicate form.

I-8.(a) delete(a,[b,c],[a,c]); this call fails.

(b) X=b

I-9. For instance, we want to convert the list

[a,b,c,d,e,f]

to the list

[[a,b],[c,d],[e,f]]

We can write the first list as [a,b|L] and the second as [[a,b]|PL]. And there's the additional condition that PL is the pairing of list L. Hence:

```
pair([],[]). pair([X,Y|L],[[X,Y]|PL]) :- pair(L,PL).
```

I-10. intersection([],S2,[]). intersection([X|S1],S2,[X|I]) :- member(X,S2), intersection(S1,S2,I).

intersection([X|S1],S2,I) :- intersection(S1,S2,I). member(X,[X|L]). member(X,[Y|L]) :- member(X,L).

I-11. It says the reverse of its list first argument is its second argument. Items are transferred one by one from the first argument to the second (initially empty) second argument, so the second argument, at the bottom level of recursion, ends up in reverse order like a stack. Then at this bottom level of recursion, the basis step binds the third argument to the current state of the second, making permanent so to speak the binding; this is then the answer. Note the program does the same thing when the first argument is bound and second unbound as when the second argument is bound and first unbound; when both arguments are bound, it verifies that one list is the reverse of the other.

I-12. Each of the three facts can in turn match the first predicate expression on the right side of the rule, giving:

bottom(north) :- not(bosses(north,U)). bottom(poindexter) :- not(bosses(poindexter,U)). bottom(hall) :- not(bosses(hall,U)).

There are no more facts, so we must handle "not"s. Only the one in the last rule above succeeds, giving the fact "bottom(hall)". That's all we can prove.

I-13.(a) Fact e(5) doesn't match anything. But f(3) matches the right side of the second rule, so c(3) is a fact. Fact c(3) gives the new rules (in order):

a(3) :- b(3). b(3) :- not(d(3)).

And nothing further can be concluded. So we handle nots. The first one not(e(Z)) fails because e(5) is a fact, but not(d(3)) succeeds. Hence b(3) succeeds, and then a(3) succeeds. If rules are never deleted, the remaining rule

b(Y) :- c(Y), not(d(Y)).

can lead to another rule

b(Y) :- c(Y).

but this doesn't lead to anything. Overall order of new facts: c(3), b(3), a(3).

(b) On the first cycle, the first rule fails, but the second succeeds and proves the fact c(3). The third and fourth rules can't be used yet, and the second cycle doesn't accomplish anything. So now we consider nots for a third cycle. There aren't any in the first two rules, and the not in the third rule fails, but the fourth rule now succeeds and proves b(3). And then on the fourth cycle a(3) is proved. That's all. Overall order of new facts is the same as part (a): c(3), b(3), a(3).

I-14.(a)

diagnosis(chicken_pox) :- rash, not(fever(high)). diagnosis(german_measles) :- rash, cold_symptom, fever(mild). diagnosis(measles) :- rash, cold_symptom, fever(high). diagnosis(mumps) :- fever(mild), sore_throat. diagnosis(scarlet_fever) :- fever(medium), sore_throat. diagnosis(cold) :- cold_symptom, fever(X). fever(high) :- temperature(T), T>102. fever(medium) :- temperature(T), T>100.5, T=<102. fever(mild) :- temperature(T), T>99, T=<100.5. cold_symptom :- sneezing. cold_symptom :- headache. cold_symptom :- sore_throat.

(b) That is, we query

diagnosis(X).

with the above rules and the four specified facts, and keep typing semicolons until we find all answers. However, we are told to take the diseases in a different order than the above, so it's like we rearrange the rules before querying to

diagnosis(measles) :- rash, cold_symptom, fever(high). diagnosis(scarlet_fever) :- fever(medium), sore_throat. diagnosis(chicken_pox) :- rash, not(fever(high)). diagnosis(mumps) :- fever(mild), sore_throat. diagnosis(german_measles) :- rash, cold_symptom, fever(mild). diagnosis(cold) :- cold_symptom, fever(X). fever(high) :- temperature(T), T>102. fever(medium) :- temperature(T), T>100.5, T=<102. fever(mild) :- temperature(T), T>99, T=<100.5. cold_symptom :- sneezing. cold_symptom :- headache. cold_symptom :- sore_throat.

Then the measles diagnosis fails (but establishes the cold_symptom conclusion); scarlet fever succeeds (after first establishing the fever(medium) conclusion); chicken pox succeeds; mumps fails; German measles fails; and cold succeeds.

(c) The sore throat fact matches predicate expressions in the scarlet_fever and last cold_symptom rules, and the new fact cold_symptom is proved. This last is now pursued, and it matches expressions in the measles and german_measles rules. Next temperature(101) is pursued, giving fact fever(medium). This fact causes first the scarlet_fever diagnosis and then the cold diagnosis to be proved.

The desire to go to the circus doesn't match anything in the rules, but the rash fact does match expressions in the measles, chicken pox, and German measles rules. We've run out of facts, so we not handle nots. In this case, there is a single not expression in the chicken pox rule, and its argument is false, so the expression can be removed, proving the chicken pox diagnosis. Nothing further can be proved.

(d) On the first cycle only the fever(medium) and cold_symptom facts are proved. On the second cycle, the diagnoses of scarlet fever and cold are proved. The third cycle doesn't prove anything, so nots are then handled. On the fourth cycle the chicken pox diagnosis succeeds.

(e) The diseases are considered in order of severity, so the first diagnosis reached is the one you should be most concerned about. That's helpful to know when using the program.

(f) Yes, asking questions of a user is important in this system, and decision lattices are good for question-asking applications. In fact, some "home medicine" books are organized like decision lattices. Decision lattices are particularly good for small computers, and a medical system like this could have a market on home computers. The variable can be easily removed by using predicates like "high_fever" and including ranges in temperature questions, as "Is the fever 99 to 100.5 degrees?"

While just what questions to ask first in the lattice are not immediately obvious, techniques exist to find them by While just what questions to ask first in the lattice are not immediately obvious, techniques exist to find them by focussing on the predicates appearing most often in rules. Just looking at the 12 rules in this problem, a good first question would be whether there's a rash, since three diagnoses mention it and three don't; the next question could be about temperature, because that helps a lot to distinguish among the diseases; then one more question will establish the most serious possible disease.

Decision lattices implicitly represent the answers to previous questions from their location in the tree, so inability to cache is not usually a major obstacle. Inability to backtrack doesn't seem important in this application.

(g) No. Clearly medical rules like these must ask questions of a user since their main application is for non-medical users (doctors have memorized these basic things), and medical facts require judgement by an observer. Computation speed is unimportant when you're asking questions and must wait for the user to answer. Also, you can't ask a user two questions simultaneously, and nearly all predicate expressions in a rule will require a question of the user. And speed of the rule-based system isn't important: medical actions rarely require action within seconds, whereas a well-implemented 1000-rule system should be able to complete its job in at most a second. So if you want a compiled form, the decision lattice sounds much better.

I-15. Or-parallelism is possible between q and u to begin; if u succeeds in binding its argument, q can be terminated. Otherwise, if q succeeds in finding an X, wait until u has failed, and then proceed to query r to bind Y, and simultaneously check s with and-parallelism. If either process fails, terminate both processes and backtrack to q to get another X. If r succeeds and s has either succeed or not finished, start verifying condition t on the same processor that bound Y. If s fails, kill both processes and start q again; if t fails, use the processor that tried to verify it to run r again. If both s and t succeed, the query succeeds.

I-16. Have "impermissible" facts that say what combinations don't go together. Then when parsing or generating a sentence, keep a record via "asserta" of what substitutions were made, and compare to this.

I-17.(a) forward chaining, since it starts from facts and works to a conclusion; it doesn't try to "confirm" anything

(b) forward chaining too, since signals coming in represent facts, which then are propagated through gates to conclusion lines

I-18. Both these questions just involve redefinition of the expected cost R of the rules.

(a) R=MQH + 2(1-MQ)H. So caching is desirable if P(2-MQ)H>1. (b) R=Q*4H/M*M + Q*2*4H/M*M + Q*3*4H/M*M + ... + (1-MQ)*4H/M or R=(2H/M)[Q+2-MQ].

I-19. The intermediate predicates (plus one diagnosis) are the only things that might be useful to cache. But so little computation is needed for any of them that caching would be useless--after all, it takes a lot more microseconds for a user to type answers to queries than for Prolog to explore all the nooks and crannies of this code. Notice there isn't any list processing in the program, a major time-waster in many Prolog programs.

I-20. Any arithmetic computations cannot be "matched" for, but should still be kept in the rule, having their variables substituted for as the variables become bound. Whenever all but arithmetic-computation expressions have been matched for on the right side of some rule, the computations should be performed, and the left side made a fact with the appropriate numerical bindings.

I-21. Note the double "not" prevents backtracking. prog([]). prog([P|L]) :- not(negcall(P)), prog(L). negcall(P) :- not(poscall(P)). poscall(P) :- call(P). poscall(P).

I-22. [specific to your elevator]

I-23. [it depends on your advertisement]

I-24. The rules look like:

russian(0.5) :- russian_maneuvers. russian(P) :- waveform(z37,P2), andcombine([0.8,P2],P).

And we're told we have the facts

russian_maneuvers. waveform(z37,0.5).

The independence assumption would give an and-combination of 0.8*0.5 = 0.4 in the second rule, and the or-combination of this with 0.5 would be 1-((1-0.5)(1-0.4)) = 0.7. That's the answer in part (a).

The conservative assumption would give max(0.8+0.5-1,0) = 0.3 in the second rule, and the or-combination of this with 0.5 is max(0.3,0.5) = 0.5. That's not the answer in part (b), but you didn't need to compute it anyway, because 1.0 is more than 0.7, and conservative combination always gives the smallest possible values.

The liberal assumption would give min(0.8,0.5) = 0.5 in the second rule, and the or-combination of this with 0.5 is min(0.5+0.5,1) = 1.0. That's the answer in part (b).

I-25. (a)

```
c(P) :- bagof(P,c2(P),PL), orcombine(PL,P). c2(P) :- a(P1), b(P2), andcombine([P1,P2],P). c2(P) :- a(P1),
b(P2), NP1 is 1-P1, NP2 is 1-P2, andcombine([NP1,NP2,0.8],P). d(P) :- a(P1), b(P2), NP2 is 1-P2,
andcombine([P1,NP2,0.5],P). e(P) :- a(P1), b(P2), NP1 is 1-P1, andcombine([NP1,P2],P).
```

And define "andcombine" and "orcombine" differently for parts (b) and (c) from parts (d) and (e).

For parts (b)-(e), it helps to define:

```
test(Pb) :- checkretract(b(X)), asserta(b(Pb)), spectrum(L), test2(L). test2([]). test2([Pa|L]) :-
checkretract(a(Y)), asserta(a(Pa)), b(Pb), c(Pc), d(Pd), e(Pe), write([Pa,Pb,Pc,Pd,Pe]), nl, test2(L).
spectrum([0.0,0.2,0.4,0.6,0.8,1.0]). checkretract(P) :- call(P), retract(P). checkretract(P) :- not(call(P)).
```

which automatically generates the results needed in a list form shown below. The Figures on the next page show parts (b)-(e).

(b)

```
?- test(0.4). [0,0.4,0.48,0,0.4] [0.2,0.4,0.433279,0.0599999,0.32] [0.4,0.4,0.401919,0.12,0.24]
[0.6,0.4,0.38592,0.18,0.16] [0.8,0.4,0.38528,0.24,0.08] [1,0.4,0.4,0.3,0]
```

(c)

```
?- test(0.9). [0,0.9,0.0800002,0,0.9] [0.2,0.9,0.23248,0.01,0.719999]
[0.4,0.9,0.39072,0.0200001,0.539999] [0.6,0.9,0.554719,0.0300001,0.36]
[0.8,0.9,0.724479,0.0400001,0.18] [1,0.9,0.9,0.0500002,0]
```

.PA .PA

(d) Now "andcombine" and "orcombine" are changed to refer to conservative combination.

```
?- test(0.4). [0,0.4,0.4,0,0.4] [0.2,0.4,0.199999,0,0.2] [0.4,0.4,0,0,0] [0.6,0.4,0,0,0] [0.8,0.4,0.2,0,0]
[1,0.4,0.4,0.0999999,0]
```

(e)

```
?- test(0.9). [0,0.9,0,0,0.9] [0.2,0.9,0.0999995,0,0.699999] [0.4,0.9,0.299999,0,0.5] [0.6,0.9,0.5,0,0.3]
[0.8,0.9,0.699999,0,0.0999999] [1,0.9,0.9,0,0]
```

(f) The curves show sharp corners, which doesn't seem too reasonable for an evidence combination method. It's not fair to cite being close to zero as a disadvantage of the conservative method, because maybe the true probabilities really are close to zero; strong correlations are not unusual between terms in expert systems, and then the conservative formula may best describe things.

I-26. (a) c

(b) b

(c) b

I-27. (a) Probably not, because you don't know what your goal states are in advance; they're just any state (object) that has a filled-in value for the property you're interested in. (Note "bidirectional" for search doesn't mean downwards and upwards inheritance, but start to goals and goals to start.)

(b) Upwards reasoning to general or bigger concepts from specific or smaller concepts would have a smaller branching factor, since there are fewer general and bigger concepts. As for start to goals reasoning vs. goals to start reasoning, assuming you could do them both, you can't reach any conclusion about which is better because goals could be either above or below the start on the inheritance hierarchy.

(c) One example: don't look for an inheritance path that is more than three links long.

(d) One example: in inheriting properties of parts of a car, try a_kind_of links before part_of links.

(e) No, because there are no obvious intermediate states where paths converge.

(f) No, because if an earlier state had a "part_of" link departing from it, that doesn't mean every later state on a path does.

I-28.

```
start(Ans) :- bagof(F,fact(F),FL), bagof(R,rule(A,B),RL), search([FL,[],RL],Ans).
successor([[F|FL],UFL,RL],[NFL,[F|UFL],RL2]) :- pursue(F,RL,RL2,NFL). pursue(F,[],[],F) :- !.
pursue(F,[[Rleft,Rright]|RL],[[Rleft,Rright]|NRL],NFL) :- not(member(F,R)), !, pursue(F,RL,NRL,NFL).
pursue(F,[Rleft,Rright]|RL],NRL,[Rleft|NRL]) :- delete(F,R,[]), pursue(F,RL,NRL,NFL).
pursue(F,[Rleft,Rright]|RL],NRL,[[Rleft,NRright]|NRL]) :- delete(F,Rright,NRright),
pursue(F,RL,NRL,NFL). eval([A,B],N) :- length(A,N1), length(B,N2), N is N1+N2. I-29. [too long to
give]
```

I-30.

```
go(Ops) :- means_ends([fire(3),closed,dry], [fire(0),dry],Ops,Finalstate). recommended([fire(X)],hose).
```

```
recommended([open],drain). recommended([dry],wait). precondition(hose,[dry,open]).
precondition(drain,[closed]). precondition(wait,[]). deletepostcondition(hose,[dry]).
addpostcondition(hose,[flooded]). deletepostcondition(drain,[closed]). addpostcondition(drain,[open]).
deletepostcondition(wait,[flooded]). addpostcondition(wait,[dry]). transform(hose,L,[fire(0)|L2]) :-
singlemember(fire(1),L), delete(fire(1),L,L2). transform(hose,L,[fire(Y)|L2]) :- singlemember(fire(X),L),
X>1, delete(fire(X),L,L2), Y is X-2. transform(wait,L,L) :- singlemember(fire(0),L).
transform(wait,L,[fire(Y)|L2]) :- singlemember(fire(X),L), X>0, delete(fire(X),L,L2), Y is X+1.
transform(drain,L,L).
```

I-31. One way:

```
/* The robot housekeeper problem by means-ends analysis */ test(A,B) :-
means_ends([lastact(none),robot(chute),basket(1,1), basket(2,2),dusty(1),dusty(2),trashy(1),untrashy(2),
notholding(1),notholding(2),vacuumed(1),vacuumable(2)], [robot(1),dusted(1),dusted(2),vacuumed(1),
vacuumed(2),basket(1,1),basket(2,2),untrashy(1), untrashy(2),notholding(1),notholding(2)],A,B).

recommended([dusted(X)],dust(X)). recommended([vacuumed(X)],vacuum(X)).
recommended([untrashy(X)],dispose(X)). recommended([holdingbasket(X)],pickup(X)).
recommended([notholding(X)],putdown(X)). recommended([basket(X,Y)],carry(X,Y)).
recommended([robot(X)],go(X)).

precondition(dust(X),[robot(X),dusty(X)]).
precondition(vacuum(X),[robot(X),vacuumable(X),dusted(X)]).
precondition(dispose(X),[robot(chute),trashy(X),
holdingbasket(X),basket(X,chute),dusted(X),vacuumed(X)]).
precondition(pickup(X),[basket(X,Y),robot(Y),trashy(X), notholding(X)]).
precondition(putdown(X),[basket(X,X),robot(X), holdingbasket(X),untrashy(X)]).
precondition(carry(X,Y),[basket(X,Z),robot(Z), holdingbasket(X)]).
precondition(go(X),[notholding(1),notholding(2)]).

deletepostcondition(dust(X),[dusty(X),vacuumed(X), lastact(A)]).
deletepostcondition(vacuum(X),[vacuumable(X),untrashy(X), lastact(A)]).
deletepostcondition(dispose(X),[trashy(X),lastact(A)]).
deletepostcondition(pickup(X),[notholding(X),lastact(A)]).
deletepostcondition(putdown(X),[holdingbasket(X), lastact(A)]).
deletepostcondition(carry(X,Y),[basket(X,Z),robot(Z), lastact(A)]).
deletepostcondition(go(X),[robot(Y),lastact(A)]).

addpostcondition(dust(X),[dusted(X),vacuumable(X), lastact(dust(X))]).
addpostcondition(vacuum(X),[vacuumed(X),trashy(X), lastact(vacuum(X))]).
addpostcondition(dispose(X),[untrashy(X), lastact(dispose(X))]).
addpostcondition(pickup(X),[holdingbasket(X), lastact(pickup(X))]).
addpostcondition(putdown(X),[notholding(X), lastact(putdown(X))]).
addpostcondition(carry(X,Y),[robot(Y),basket(X,Y), lastact(carry(X,Y))]).
addpostcondition(go(X),[robot(X),lastact(go(X))]).
```

Here is it running:

```
?- test(A,B). A=[go(1),dust(1),go(2),dust(2),go(1),vacuum(1),go(2),
vacuum(2),go(1),pickup(1),carry(1,chute),dispose(1),
carry(1,1),putdown(1),go(2),pickup(2),carry(2,chute), dispose(2),carry(2,2),putdown(2),go(1)]
B=[robot(1),lastact(go(1)),notholding(2),basket(2,2), untrashy(2),notholding(1),basket(1,1),untrashy(1),
vacuumed(2),vacuumed(1),dusted(2),dusted(1)]
```

In one version of Prolog, this program used 45K of heap storage, 31K of global stack, and 41K of local stack, and took 3.9 seconds. That's a lot better than the A* version of the same program in Exercise 10-9, but the above answer isn't optimal.

I-32. This means that goal descriptions will also need to contain "not" expressions. So we must add one more rule to the definition of the "difference" predicate:

```
difference([],S,[]). difference([not(P)|G],S,G2) :- not(singlemember(P,S)), !, difference(G,S,G2).
difference([P|G],S,G2) :- singlemember(P,S), !, difference(G,S,G2). difference([P|G],S,[P|G2]) :-
difference(G,S,G2).
```

(In some Prolog dialects you must write "not P" above rather than "not(P)".) You'll also probably want to modify the "recommended" rules to exploit these "not" terms.

I-33. There are no single-variable constraints. Picking variable X first, the first and third expressions both give the possibility list [1,2,3,5]. Picking variable Y next, value 4 is possible by the first expression, but not by the second; value 5 is possible by both; value 3 is possible by both; value 6 is possible by the first but not the second; hence the list [3,5]. Picking variable Z next, 3 satisfies all expressions; 6 fails the third expression; 4 fails the fourth expression. So Z must be 3. Returning to X, it must be 3 by the third expression. And Y must be 3 by the second expression.

I-34. Just subdivide the duplication-checking in the query into checking for each of ten pairs. Move the pair checking to right after the pairs values are bound:

```
test([T1,T2,T3,T4,T5]) :- classtime(T1), not(occupied(T1)), classtime(T2), not(occupied(T2)),
not(T1=T2), classtime(T3), not(occupied(T3)), not(T3=T1), not(T3=T2), classtime(T4),
not(occupied(T4)), not(T4=T1), not(T4=T2), not(T4=T3), classtime(T5), not(occupied(T5)), not(T5=T1),
not(T5=T2), not(T5=T3), not(T5=T4), not(two_consecutive_hours([T1,T2,T3,T4,T5])),
not(three_classes_same_day([T1,T2,T3,T4,T5])).
```

I-35. (a) (ii)

(b) (i)

I-36.(a) possibility lists

(b) lists of probabilities for combination

(c) the search agenda or the paths found to states

(d) the places to backtrack to or the pointers to the database for each previous predicate expression

I-37.(a) yes, if the operator definitions can create a cycle of branches; this is a common problem of depth-first

(b) no, costs would have to keep increasing for any cycle until they were more than the solution path

(c) yes, subtle ordering problems can create loops (remember the means-ends discussed in the book is depth-first)

(d) no, each cycle must cross something out or the cycling stops, and there are a finite number of things to cross out

(e) no, there can't be cycles involving "part_of" because if X is part of Y, then Y can't be part of X, or else the "part_of" concept wouldn't be useful (or else there's a user mistake)

(f) yes, since "is_another_name_for" can make cycles

(g) no, since repeated actions or duplicate clauses are not allowed by any sensible resolution strategy

I-38.(a)

c(a,p,b). c(a,q,c). c(b,q,e). c(b,r,d). c(c,r,d). c(d,r,f). c(e,r,f).

(b) defective(O2) :- c(O1,P,O2), faulty(P). defective(O2) :- c(O1,P,O2), defective(O1).

(c) The query is

?- defective(d).

We can try the first rule, but it fails since process "r" isn't faulty. We can try the second rule, binding O2=d, O1=b, and P=r. Then the new query is

?- defective(b).

And this can match the first rule with O2=b, O1=a, and P=p, since we know that p is faulty. So the b query succeeds, and hence the "d" query succeeds.

(d) defective(O2); not(c(O1,P,O2)); not(faulty(P)). defective(O2); not(c(O1,P,O2)); not(defective(O1)).

(e) Obviously we have to reason backward from f through r through d through r again to c. The "faulty" clause obviously can't help since the given and the conclusion are both about defective objects. So we just use the second clause in part (d) twice. Resolve it first with the fact not(defective(f)) to get

not(c(O1,P,f)); not(defective(O1)).

Resolve this with the fact c(d,r,f) to get

not(defective(d)).

as is obvious from the diagram. Now resolve this fact with again the second clause in part (d):

not(c(O1,P,d)); not(defective(O1)).

And resolve that with the fact c(c,r,d) to get

not(defective(c)).

(f) Yes if we use something other than depth-first and if we suspect that X and Y are connected, since the goal state is known and is unique, and the branching factor is similar in both directions. But if it's more likely that X and Y aren't connected, then the bidirectional search could require significantly more work than a unidirectional search.

(g) (2+2+1+1+1) / 6 = 7/6 (h) successor(X,NX) :- c(X,P,NX).

(i) Designate a variable for each object and each process (the letters in the diagram for part (a)). The values for object variables are "defective" and "nondefective"; the values for process variables are "faulty" and "nonfaulty". The constraints are the connection facts for part (a) and the rules for part (b). But there's a subtle point: the rules can be used backwards as well as forwards, so there should really be four Prolog-style rules total.

(j) Slot inheritance examples: input, output, number of inputs, number of outputs, process name, process type. Value inheritance examples: number of inputs, number of outputs, process name, process type.

(k) Inherit the value from the "a_kind_of" superframe. The "a_kind_of" relationship is usually more fundamental, and in this case the "part_of" superframe can include a lot of irrelevant processes.

(l) Some possibilities:

--Draw the data-flow diagram and assign Cartesian coordinates to the objects; the evaluation function is the straight-line distance between states in the diagram.

--Designate all no-input objects as level 0; then assign level numbers to every other object in the diagram as in breadth-first search. If there are two or more level numbers for a state, pick the smallest. Then the evaluation function is the difference between level numbers of start and goal.

--Group objects into modules, and create a graph indicating which modules connect directly to which other modules. Let M be the maximum number of nodes in a module. Then an evaluation function is M times the number of links from the module containing X to the module containing Y.

--Let the evaluation function be 1 for objects that are not defective, 0 for objects that are. This is a weak help, but it does qualify as an evaluation function.

Go to book index

Errata for Neil Rowe, Artificial Intelligence Through Prolog
February 24, 1989
Please send any other errata you find to the author at code CS/Rp, Naval Postgraduate School, Monterey CA 93943; or send email to rowe@cs.nps.navy.mil.

(* = Prentice-Hall not yet informed)

Errata for book itself:

Page xvi, end of last sentence: change "nps-cs.arpa" to "cs.nps.navy.mil".

Page 12, second paragraph of section 2.7: change "circle" to "oval" everywhere it occurs.

Page 13, 5th line from bottom: replace "Here 'to be' is used" with "Here forms of the verb "to be" are used"

Page 31, figure 3-3: arrow missing from "X = tom, Y = dick" to "Z = harry"

Page 44, 19th line: change "part_of" to "a_kind_of" so the whole line should read "color_enterprise(C) :- a_kind_of(enterprise,X), color(X,C)."

Page 77, first line: change "1" (numeral one) to "0" (numeral zero).

Page 87, section 5.7, sixth line: change the second right bracket to a right parenthesis, so it reads:

```
subset([X|L1],L2) :- member(X,L2), subset(L1,L2).
```

Page 103, seventh and sixth lines from the end of section 6.2: change "old" to "new", and change "new" to "old" in the sentence "It is redundant if the old rule always succeeds whenever the new rule succeeds."

Page 105, figure 6-2, step 7, end of line: replace "c." with "c(2)." (note the period must remain on the end)

(*) Page 107, Figure 6-4: "R6" column should read top to bottom in order: "fails, -, -, -, -" rather than "fails, fails, fails, fails, -".

Page 129, 7th line from bottom: insert an apostrophe just before "Are"

Page 132, 3rd line, fourth word: "emdry" should be just "dry"

(*) Page 140, insert before 9th line sentence beginning "When forward chaining..." : "This program does not delete redundant rules like the algorithm in section 6.2."

(*) Page 149, indented material starting "forward :- done.": replace all but the first three lines, by the following:

```
pursuit(F) :- rules(F,Rlist), append(RlistF,[[L,R]|RlistB],Rlist),
  rule_pursuit(F,L,R,RlistF,RlistB).
rule_pursuit(F,L,R,RlistF,RlistB) :- member(F,R), delete(F,R,Rnew),
  retract(rules(F,Oldrules)), append(RlistF,RlistB,NewRlist),
  new_rule(F,L,Rnew,NewRlist).
new_rule(F,L,[],Rlist) :- asserta(rules(F,Rlist)), not(fact(L)),
  asserta(fact(L)).
new_rule(F,L,R,Rlist) :- not(empty_list(R)), asserta(rules(F,[[L,R]|Rlist])).
empty_list([]).
append([],L,L).
append([X|L],L2,[X|L3]) :- append(L,L2,L3).
```

Page 188, 6th line from bottom: "battery" should be "lottery"

Page 196, Figure 9-3: the "remove screws" in the lower left should be "remove cables"

(*) Page 203, 19th line from bottom: change "certain" to "restricted"

Page 203, 11th line from bottom: change "A to S" to "S to A"

Page 203, 11th line from bottom: change "3" in "A to 3" to "A to E"

Page 221, second of the three groups of three indented lines: in all three cases the period must occur after the right parentheses, not before it; in the hyphen in "data-point" should be an underscore symbol.

(*) Page 265, Figure 11-1, 5th column of table: delete "?" after "visit"

Page 277, Figure 11-7, "Level 2" box on left, sixth line: delete right bracket on the end, just after the comma

Page 289, Figure 12-1: the "frame_name: memos to Ann" box should have a separate line with an arrow on it going to the box labeled "frame_name: memo", and this arrow cannot be combined with the one from frame_name: budget memos" to "frame_name: memo", which must have its own line with an arrow on it.

Page 321, line 14: insert "P and" before "all the expressions to the left".

Page 362, Figure 14-4: the small upper center triangle-like region that is cross-hatched should not be; it should be just have lines crossing it northwest to southeast, like the region to the left of it.

Page 363, problem 14-8: in fourth line, put parentheses around X and 1; in sixth line, put parentheses around N. In all cases, no space after the "p".

Page 383, 12th line from the bottom: delete "the" from "an included variable X, then that the expression".

Page 388, item 6, second line: delete "unbound" (so same word won't occur twice in succession)

Page 390, sixth line from the bottom: change "*lattice* or *directed acycylic graph*" to "*directed acyclic graph* and usually also a *lattice*".

Page 407, problem 2-6, 10th line: **ship** should be **ship_info**

Page 409, fifth line: the "4-5" is on the wrong line: it should be on the left side of the sixth line in front of the word "No". The rest of the fifth line is correct, however; the "(b)" should be the first thing on the line.

Page 409, problem 4-7, part (e), second line: change "Dick" to "Sue"

Page 415, 10th line of text from bottom: "L" should be "Z", so the line should read "?- mystery([a,b,c,d],Z)." now.

Page 416, problem 5-22, 6th line: change "not(member(X,L),L)" to "not(member(X,L))".

Page 417, problem 6-1(a), 6th and 7th line from top: change the sentence "This in turn matches an expression in R2." to "This doesn't match anything (R2 is being ignored)."

Page 417, problem 6-5(b), item "1": replace by: "1. The first and third rules fail because they mention at least one thing that doesn't match a fact, and the second rule is skipped."

Page 418, problem 6-5(b), item "5": replace by: "5. The first rule again fails, and the second rule is again skipped."

Errata for Instructors' Manual

Page 22: Add another arrow from "C" circle to the left side of the left most "R" box.

Page 26: The "part_of" arrows are backwards.

Page 96, line 3: delete duplicate "extra"

Page 96, line 9: change "rule" to "ruling"

Page 103: The point at (.3, .8) should be at (.2, .8).

Page 110: This figure is for Problem 12-7.

Go to book index