PROLOG PROGRAMMING FOR ARTIFICIAL INTELLIGENCE

INTERNATIONAL COMPUTER SCIENCE SERIES

Consulting editors **A D McGettrick** University of Strathclyde
J van Leeuwen University of Utrecht

OTHER TITLES IN THE SERIES

Programming in Ada (2nd Edn.) *J G P Barnes*

Computer Science Applied to Business Systems *M J R Shave and K N Bhaskar*

Software Engineering (2nd Edn.) *I Sommerville*

A Structured Approach to FORTRAN 77 Programming *T M R Ellis*

The Cambridge Distributed Computing System *R M Needham and A J Herbert*

An Introduction to Numerical Methods with Pascal *L V Atkinson and P J Harley*

The UNIX System *S R Bourne*

Handbook of Algorithms and Data Structures *G H Gonnet*

Office Automation: Concepts, Technologies and Issues *R A Hirschheim*

Microcomputers in Engineering and Science *J F Craine and G R Martin*

UNIX for Super-Users *E Foxley*

Software Specification Techniques *N Gehani and A D McGettrick* (eds.)

Introduction to Expert Systems *P Jackson*

Data Communications for Programmers *M Purser*

Local Area Network Design *A Hopper, S Temple and R C Williamson*

Modula-2: Discipline & Design *A H J Sale*

PROLOG PROGRAMMING FOR ARTIFICIAL INTELLIGENCE

Ivan Bratko

E. Kardelj University · J. Stefan Institute
Yugoslavia



ADDISON-WESLEY
PUBLISHING
COMPANY

Wokingham, England · Reading, Massachusetts · Menlo Park, California
Don Mills, Ontario · Amsterdam · Sydney · Singapore · Tokyo
Madrid · Bogota · Santiago · San Juan

© 1986 Addison-Wesley Publishers Limited
© 1986 Addison-Wesley Publishing Company, Inc.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without prior written permission of the publisher.

Cover graphic by kind permission of Dicomed (UK) Ltd.
Phototypeset by Computerset (MFK) Ltd., Ely, Cambs.
Printed in Great Britain by Commercial Colour Press.

British Library Cataloguing in Publication Data
Bratko, Ivan
Prolog programming for artificial intelligence.
1. Artificial intelligence—Data processing
2. Prolog (Computer program language)
I. Title
006.3'02855133 Q336

ISBN 0-201-14224-4

Library of Congress Cataloging-in-Publication Data
Bratko, Ivan.
Prolog programming for artificial intelligence.

Includes index.
1. Artificial intelligence—Data processing.
2. Prolog (Computer program language) I. Title.
Q336.B74 1986 006.3 86-1092
ISBN 0-201-14224-4

ABCDEF 89876

To Branka, Andrej and Tadej

Foreword

In the Middle Ages, knowledge of Latin and Greek was essential for all scholars. The one-language scholar was necessarily a handicapped scholar who lacked the perception that comes from seeing the world from two points of view. Similarly, today's practitioner of Artificial Intelligence is handicapped unless thoroughly familiar with both Lisp and Prolog, for knowledge of the two principal languages of Artificial Intelligence is essential for a broad point of view.

I am dedicated to Lisp, having grown up at MIT where Lisp was invented. Nevertheless, I can never forget my excitement when I saw my first Prolog-style program in action. It was part of Terry Winograd's famous Shrdlu system, whose blocks-world problem solver arranged for a simulated robot arm to move blocks around a screen, solving intricate problems in response to human-specified goals.

Winograd's blocks-world problem solver was written in Microplanner, a language which we now recognize as a sort of Prolog. Nevertheless, in spite of the defects of Microplanner, the blocks-world problem solver was organized explicitly around goals, because a Prolog-style language encourages programmers to think in terms of goals. The goal-oriented procedures for grasping, clearing, getting rid of, moving, and ungrasping made it possible for a clear, transparent, concise program to seem amazingly intelligent.

Winograd's blocks-world problem solver permanently changed the way I think about programs. I even rewrote the blocks-world problem solver in Lisp for my Lisp textbook because that program unalterably impressed me with the power of the goal-oriented philosophy of programming and the fun of writing goal-oriented programs.

But learning about goal-oriented programming through Lisp programs is like reading Shakespeare in a language other than English. Some of the beauty comes through, but not as powerfully as in the original. Similarly, the best way to learn about goal-oriented programming is to read and write goal-oriented programs in Prolog, for goal-oriented programming is what Prolog is all about.

In broader terms, the evolution of computer languages is an evolution away from low-level languages, in which the programmer specifies *how* something is to be done, toward high-level languages, in which the programmer specifies simply *what* is to be done. With the development of Fortran, for example, programmers were no longer forced to speak to the computer in the procrustian low-level language of addresses and registers. Instead, Fortran

programmers could speak in their own language, or nearly so, using a notation that made only moderate concessions to the one-dimensional, 80-column world.

Fortran and nearly all other languages are still how-type languages, however. In my view, modern Lisp is the champion of these languages, for Lisp in its Common Lisp form is enormously expressive, but how to do something is still what the Lisp programmer is allowed to be expressive about. Prolog, on the other hand, is a language that clearly breaks away from the how-type languages, encouraging the programmer to describe situations and problems, not the detailed means by which the problems are to be solved.

Consequently, an introduction to Prolog is important for all students of Computer Science, for there is no better way to see what the notion of what-type programming is all about.

In particular, the chapters of this book clearly illustrate the difference between how-type and what-type thinking. In the first chapter, for example, the difference is illustrated through problems dealing with family relations. The Prolog programmer straightforwardly describes the grandfather concept in explicit, natural terms: a grandfather is a father of a parent. Here is the Prolog notation:

```
grandfather( X, Z)  :-  father( X, Y), parent( Y, Z).
```

Once Prolog knows what a grandfather is, it is easy to ask a question: who are Patrick's grandfathers, for example. Here again is the Prolog notation, along with a typical answer:

```
?-  grandfather( X, patrick).

X = james;

X = carl
```

It is Prolog's job to figure out how to solve the problem by combing through a database of known father and parent relations. The programmer specifies only what is known and what question is to be solved. The programmer is more concerned with knowledge and less concerned with algorithms that exploit the knowledge.

Given that it is important to learn Prolog, the next question is how. I believe that learning a programming language is like learning a natural language in many ways. For example, a reference manual is helpful in learning a programming language, just as a dictionary is helpful in learning a natural language. But no one learns a natural language with only a dictionary, for the words are only part of what must be learned. The student of a natural language must learn the conventions that govern how the words are put legally together, and later, the student should learn the art of those who put the words together with style.

Similarly, no one learns a programming language from only a reference

manual, for a reference manual says little or nothing about the way the primitives of the language are put to use by those who use the language well. For this, a textbook is required, and the best textbooks offer copious examples, for good examples are distilled experience, and it is principally through experience that we learn.

In this book, the first example is on the first page, and the remaining pages constitute an example cornucopia, pouring forth Prolog programs written by a passionate Prolog programmer who is dedicated to the Prolog point of view. By carefully studying these examples, the reader acquires not only the mechanics of the language, but also a personal collection of precedents, ready to be taken apart, adapted, and reassembled together into new programs. With this acquisition of precedent knowledge, the transition from novice to skilled programmer is already under way.

Of course, a beneficial side effect of good programming examples is that they expose a bit of interesting science as well as a lot about programming itself. The science behind the examples in this book is Artificial Intelligence. The reader learns about such problem-solving ideas as problem reduction, forward and backward chaining, 'how' and 'why' questioning, and various search techniques.

In fact, one of the great features of Prolog is that it is simple enough for students in introductory Artificial Intelligence subjects to learn to use immediately. I expect that many instructors will use this book as part of their artificial-intelligence subjects so that their students can see abstract ideas immediately reduced to concrete, motivating form.

Among Prolog texts, I expect this book to be particularly popular, not only because of its examples, but also because of a number of other features:

- Careful summaries appear throughout.
- Numerous exercises reinforce all concepts.
- Structure selectors introduce the notion of data abstraction.
- Explicit discussions of programming style and technique occupy an entire chapter.
- There is honest attention to the problems to be faced in Prolog programming, as well as the joys.

Features like this make this a well done, enjoyable, and instructive book.

Patrick H. Winston
Cambridge, Massachusetts
January 1986

Preface

Prolog is a programming language centred around a small set of basic mechanisms, including pattern matching, tree-based data structuring, and automatic backtracking. This small set constitutes a surprisingly powerful and flexible programming framework. Prolog is especially well suited for problems that involve objects – in particular, structured objects – and relations between them. For example, it is an easy exercise in Prolog to express the spatial relationships suggested in the cover illustration – such as, the top sphere is behind the left one. It is also easy to state a more general rule: if X is closer to the observer than Y and Y is closer than Z, then X must be closer than Z. Prolog can now reason about the spatial relations and their consistency with respect to the general rule. Features like this make Prolog a powerful language for Artificial Intelligence and non-numerical programming in general.

Prolog stands for *programming in logic* – an idea that emerged in the early 1970s to use logic as a programming language. The early developers of this idea included Robert Kowalski at Edinburgh (on the theoretical side), Maarten van Emden at Edinburgh (experimental demonstration), and Alain Colmerauer at Marseilles (implementation). The present popularity of Prolog is largely due to David Warren's efficient implementation at Edinburgh in the mid 1970s.

Since Prolog has its roots in mathematical logic it is often introduced through logic. However, such a mathematically intensive introduction is not very useful if the aim is to teach Prolog as a practical programming tool. Therefore this book is not concerned with the mathematical aspects, but concentrates on the art of making the few basic mechanisms of Prolog solve interesting problems. Whereas conventional languages are procedurally oriented, Prolog introduces the descriptive, or *declarative*, view. This greatly alters the way of thinking about problems and makes learning to program in Prolog an exciting intellectual challenge.

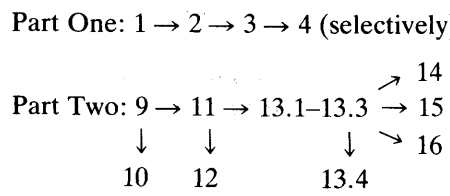
Part One of the book introduces the Prolog language and shows how Prolog programs are developed. Part Two demonstrates the power of Prolog applied in some central areas of Artificial Intelligence, including problem solving and heuristic search, expert systems, game playing and pattern-directed systems. Fundamental AI techniques are introduced and developed in depth towards their implementation in Prolog, resulting in complete programs. These can be used as building blocks for sophisticated applications. Techniques to handle important data structures, such as trees and graphs, are also included

although they do not strictly belong to AI. These techniques are often used in AI programs and their implementation helps to learn the general skills of Prolog programming. Throughout, the emphasis is on the clarity of programs; efficiency tricks that rely on implementation-dependent features are avoided.

This book is for students of Prolog and Artificial Intelligence. It can be used in a Prolog course or in an AI course in which the principles of AI are brought to life through Prolog. The reader is assumed to have a basic general knowledge of computers, but no knowledge of AI is necessary. No particular programming experience is required; in fact, plentiful experience and devotion to conventional procedural programming, for example in Pascal, might even be an impediment to the fresh way of thinking Prolog requires.

Among several Prolog dialects, the Edinburgh syntax, also known as DEC-10 syntax, is the most widespread, and is therefore also adopted in this book. For compatibility with the various Prolog implementations, this book only uses a relatively small subset of the built-in features that are shared by many Prologs.

How to read the book? In Part One, the natural reading order corresponds to the order in the book. However, the part of Section 2.4 that describes the procedural meaning of Prolog in a more formalized way can be skipped. Chapter 4 presents programming examples that can be read (or skipped) selectively. Part Two allows more flexible reading strategies as the chapters are intended to be mutually independent. However, some topics will still be naturally done before others – for example, the basics of data structures (Chapter 9) and basic search strategies (Chapters 11 and 13). The following diagram summarizes the constraints on natural reading sequences:



There are some controversial views that historically accompanied Prolog. Prolog has fast gained popularity in Europe as a practical programming tool. In Japan, Prolog was placed at the centre of the development of the Fifth Generation computers. On the other hand, in the United States its acceptance began with some delay, due to several historical factors. One of these originated from a previous American experience with the Microplanner language, also akin to the idea of logic programming, but inefficiently implemented. This early experience with Microplanner was unjustifiably generalized to Prolog, but was later convincingly rectified by David Warren's efficient implementation of Prolog. Reservations against Prolog also came in reaction to the 'orthodox school' of logic programming, which insisted on the use of pure logic that should not be marred by adding practical facilities not related to logic. This uncompromising position was modified by Prolog practitioners who adopted a more pragmatic view, benefiting from combining both the declarative

approach with the traditional, procedural one. A third factor that delayed the acceptance of Prolog was that for a long time Lisp had no serious competition among languages for AI. In research centres with strong Lisp tradition, there was therefore a natural resistance to Prolog. The dilemma of Prolog vs. Lisp has softened over the years and many now believe in a combination of ideas from both worlds.

Acknowledgements

Donald Michie was responsible for first inducing my interest in Prolog. I am grateful to Lawrence Byrd, Fernando Pereira and David H. D. Warren, once members of the Prolog development team at Edinburgh, for their programming advice and numerous discussions. The book greatly benefited from comments and suggestions of Andrew McGettrick and Patrick H. Winston. Other people who read parts of the manuscript and contributed significant comments include: Igor Kononenko, Tanja Majaron, Igor Mozetic, Timothy B. Niblett and Franc Zerdin. I would also like to thank Debra Myson-Etherington and Simon Plumtree of Addison-Wesley for their work in the process of making this book. Finally, this book would not be possible without the stimulating creativity of the international logic programming community.

Ivan Bratko
The Turing Institute, Glasgow
January 1986

CONTENTS

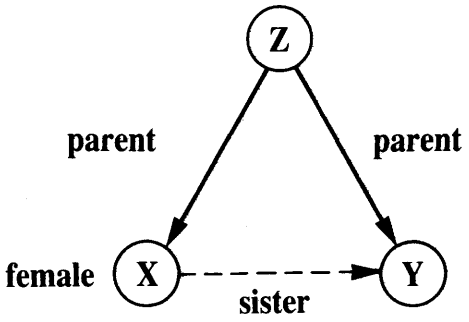
Foreword		vii
Preface		xi
PART ONE	**THE PROLOG LANGUAGE**	1
Chapter 1	**An Overview of Prolog**	3
1.1	An example program: defining family relations	3
1.2	Extending the example program by rules	8
1.3	A recursive rule definition	14
1.4	How Prolog answers questions	19
1.5	Declarative and procedural meaning of programs	24
Chapter 2	**Syntax and Meaning of Prolog Programs**	27
2.1	Data objects	27
2.2	Matching	35
2.3	Declarative meaning of Prolog programs	40
2.4	Procedural meaning	43
2.5	Example: monkey and banana	49
2.6	Order of clauses and goals	53
2.7	Remarks on the relation between Prolog and logic	60
Chapter 3	**Lists, Operators, Arithmetic**	64
3.1	Representation of lists	64
3.2	Some operations on lists	67
3.3	Operator notation	78
3.4	Arithmetic	84
Chapter 4	**Using Structures: Example Programs**	93
4.1	Retrieving structured information from a database	93
4.2	Doing data abstraction	97
4.3	Simulating a non-deterministic automaton	99
4.4	Travel planning	103
4.5	The eight queens problem	108

Chapter 5	**Controlling Backtracking**	120
5.1	Preventing backtracking	120
5.2	Examples using cut	125
5.3	Negation as failure	129
5.4	Problems with cut and negation	133
Chapter 6	**Input and Output**	137
6.1	Communication with files	137
6.2	Processing files of terms	140
6.3	Manipulating characters	147
6.4	Constructing and decomposing atoms	149
6.5	Reading programs: **consult**, **reconsult**	152
Chapter 7	**More Built-in Procedures**	155
7.1	Testing the type of terms	155
7.2	Constructing and decomposing terms: =.., **arg**, **name**	163
7.3	Various kinds of equality	168
7.4	Database manipulation	169
7.5	Control facilities	174
7.6	**bagof**, **setof** and **findall**	175
Chapter 8	**Programming Style and Technique**	179
8.1	General principles of good programming	179
8.2	How to think about Prolog programs	181
8.3	Programming style	184
8.4	Debugging	187
8.5	Efficiency	188
PART TWO	**PROLOG IN ARTIFICIAL INTELLIGENCE**	201
Chapter 9	**Operations on Data Structures**	203
9.1	Representing and sorting lists	203
9.2	Representing sets by binary trees	211
9.3	Insertion and deletion in a binary dictionary	217
9.4	Displaying trees	222
9.5	Graphs	224
Chapter 10	**Advanced Tree Representations**	233
10.1	The 2-3 dictionary	233
10.2	AVL-tree: an approximately balanced tree	241
Chapter 11	**Basic Problem-Solving Strategies**	246
11.1	Introductory concepts and examples	246
11.2	Depth-first search strategy	251
11.3	Breadth-first search strategy	256

11.4	Comments on searching graphs, on optimality, and on search complexity	262
Chapter 12	**Best-first: A Heuristic Search Principle**	265
12.1	Best-first search	265
12.2	Best-first search applied to the eight puzzle	273
12.3	Best-first search applied to scheduling	278
Chapter 13	**Problem Reduction and AND/OR Graphs**	286
13.1	AND/OR graph representation of problems	286
13.2	Examples of AND/OR representation	290
13.3	Basic AND/OR search procedures	294
13.4	Best-first AND/OR search	299
Chapter 14	**Expert Systems**	314
14.1	Functions of an expert system	314
14.2	Main structure of an expert system	315
14.3	*If-then* rules for representing knowledge	316
14.4	Developing the shell	323
14.5	Implementation	329
14.6	Dealing with uncertainty	347
14.7	Concluding remarks	355
Chapter 15	**Game Playing**	359
15.1	Two-person, perfect-information games	359
15.2	The minimax principle	361
15.3	The alpha-beta algorithm: an efficient implementation of minimax	364
15.4	Minimax-based programs: refinements and limitations	368
15.5	Pattern knowledge and the mechanism of 'advice'	370
15.6	A chess endgame program in Advice Language 0	373
Chapter 16	**Pattern-Directed Programming**	390
16.1	Pattern-directed architecture	390
16.2	A simple interpreter for pattern-directed programs	394
16.3	A simple theorem prover	396
16.4	Concluding remarks	402
Solutions to Selected Exercises		405
Index		419

PART ONE

THE PROLOG LANGUAGE



1 An Overview of Prolog

This chapter reviews basic mechanisms of Prolog through an example program. Although the treatment is largely informal many important concepts are introduced.

1.1 An example program: defining family relations

Prolog is a programming language for symbolic, non-numeric computation. It is specially well suited for solving problems that involve objects and relations between objects. Figure 1.1 shows an example: a family relation. The fact that Tom is a parent of Bob can be written in Prolog as:

parent(tom, bob).

Here we choose **parent** as the name of a relation; **tom** and **bob** are its argu-

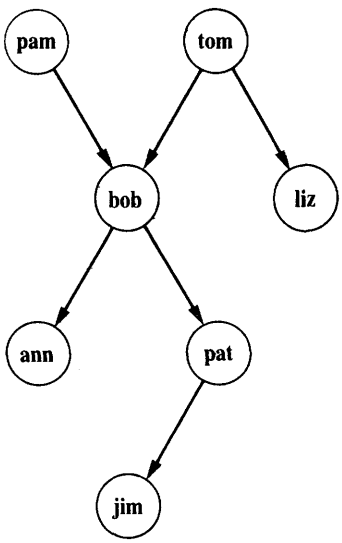


Figure 1.1 A family tree.

ments. For reasons that will become clear later we write names like **tom** with an initial lower-case letter. The whole family tree of Figure 1.1 is defined by the following Prolog program:

```
parent( pam, bob).
parent( tom, bob).
parent( tom, liz).
parent( bob, ann).
parent( bob, pat).
parent( pat, jim).
```

This program consists of six *clauses*. Each of these clauses declares one fact about the **parent** relation.

When this program has been communicated to the Prolog system, Prolog can be posed some questions about the **parent** relation. For example, Is Bob a parent of Pat? This question can be communicated to the Prolog system by typing into the terminal:

```
?- parent( bob, pat).
```

Having found this as an asserted fact in the program, Prolog will answer:

```
yes
```

A further query can be:

```
?- parent( liz, pat).
```

Prolog answers

```
no
```

because the program does not mention anything about Liz being a parent of Pat. It also answers 'no' to the question

```
?- parent( tom, ben).
```

because the program has not even heard of the name Ben.

More interesting questions can also be asked. For example: Who is Liz's parent?

```
?- parent( X, liz).
```

Prolog's answer will not be just 'yes' or 'no' this time. Prolog will tell us what is the (yet unknown) value of X such that the above statement is true. So the

answer is:

X = tom

The question Who are Bob's children? can be communicated to Prolog as:

?- parent(bob, X).

This time there is more than just one possible answer. Prolog first answers with one solution:

X = ann

We may now want to see other solutions. We can say that to Prolog (in most Prolog implementations by typing a semicolon), and Prolog will find other answers:

X = pat

If we request more solutions again, Prolog will answer 'no' because all the solutions have been exhausted.

Our program can be asked an even broader question: Who is a parent of whom? Another formulation of this question is:

Find X and Y such that X is a parent of Y.

This is expressed in Prolog by:

?- parent(X, Y).

Prolog now finds all the parent-child pairs one after another. The solutions will be displayed one at a time as long as we tell Prolog we want more solutions, until all the solutions have been found. The answers are output as:

X = pam
Y = bob;

X = tom
Y = bob;

X = tom
Y = liz;

...

We can stop the stream of solutions by typing, for example, a period instead of a semicolon (this depends on the implementation of Prolog).

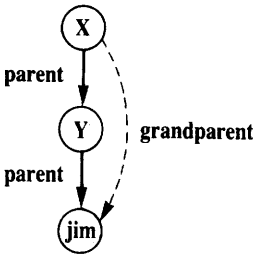


Figure 1.2 The **grandparent** relation expressed as a composition of two **parent** relations.

Our example program can be asked still more complicated questions like: Who is a grandparent of Jim? As our program does not directly know the **grandparent** relation this query has to be broken down into two steps, as illustrated by Figure 1.2.

(1) Who is a parent of Jim? Assume that this is some Y.
(2) Who is a parent of Y? Assume that this is some X.

Such a composed query is written in Prolog as a sequence of two simple ones:

```
?- parent( Y, jim), parent( X, Y).
```

The answer will be:

```
X = bob
Y = pat
```

Our composed query can be read: Find such X and Y that satisfy the following two requirements:

parent(Y, jim) and **parent(X, Y)**

If we change the order of the two requirements the logical meaning remains the same:

parent(X, Y) and **parent(Y, jim)**

We can indeed do this in our Prolog program and the query

```
?- parent( X, Y), parent( Y, jim).
```

will produce the same result.

In a similar way we can ask: Who are Tom's grandchildren?

```
?- parent( tom, X), parent( X, Y).
```

Prolog's answers are:

```
X = bob
Y = ann;

X = bob
Y = pat
```

Yet another question could be: Do Ann and Pat have a common parent? This can be expressed again in two steps:

(1) Who is a parent, X, of Ann?
(2) Is (this same) X a parent of Pat?

The corresponding question to Prolog is then:

```
?- parent( X, ann), parent( X, pat).
```

The answer is:

```
X = bob
```

Our example program has helped to illustrate some important points:

- It is easy in Prolog to define a relation, such as the **parent** relation, by stating the n-tuples of objects that satisfy the relation.
- The user can easily query the Prolog system about relations defined in the program.
- A Prolog program consists of *clauses*. Each clause terminates with a full stop.
- The arguments of relations can (among other things) be: concrete objects, or constants (such as **tom** and **ann**), or general objects such as X and Y. Objects of the first kind in our program are called *atoms*. Objects of the second kind are called *variables*.
- Questions to the system consist of one or more *goals*. A sequence of goals, such as

  parent(X, ann), parent(X, pat)

  means the conjunction of the goals:

  X is a parent of Ann, *and*
  X is a parent of Pat.

  The word 'goals' is used because Prolog accepts questions as goals that are to be satisfied.
- An answer to a question can be either positive or negative, depending on

whether the corresponding goal can be satisfied or not. In the case of a positive answer we say that the corresponding goal was *satisfiable* and that the goal *succeeded*. Otherwise the goal was *unsatisfiable* and it *failed*.

- If several answers satisfy the question then Prolog will find as many of them as desired by the user.

Exercises

1.1 Assuming the **parent** relation as defined in this section (see Figure 1.1), what will be Prolog's answers to the following questions?

(a) ?- **parent(jim, X).**

(b) ?- **parent(X, jim).**

(c) ?- **parent(pam, X), parent(X, pat).**

(d) ?- **parent(pam, X), parent(X, Y), parent(Y, jim).**

1.2 Formulate in Prolog the following questions about the **parent** relation:

(a) Who is Pat's parent?

(b) Does Liz have a child?

(c) Who is Pat's grandparent?

1.2 Extending the example program by rules

Our example program can be easily extended in many interesting ways. Let us first add the information on the sex of the people that occur in the **parent** relation. This can be done by simply adding the following facts to our program:

```
female( pam).
male( tom).
male( bob).
female( liz).
female( pat).
female( ann).
male( jim).
```

The relations introduced here are **male** and **female**. These relations are unary (or one-place) relations. A binary relation like **parent** defines a relation between *pairs* of objects; on the other hand, unary relations can be used to declare simple yes/no properties of objects. The first unary clause above can be read: Pam is a female. We could convey the same information declared in the two unary relations with one binary relation, **sex**, instead. An alternative piece

of program would then be:

```
sex( pam, feminine).
sex( tom, masculine).
sex( bob, masculine).
...
```

As our next extension to the program let us introduce the **offspring** relation as the inverse of the **parent** relation. We could define **offspring** in a similar way as the **parent** relation; that is, by simply providing a list of simple facts about the **offspring** relation, each fact mentioning one pair of people such that one is an offspring of the other. For example:

```
offspring( liz, tom).
```

However, the **offspring** relation can be defined much more elegantly by making use of the fact that it is the inverse of **parent**, and that **parent** has already been defined. This alternative way can be based on the following logical statement:

For all X and Y,
  Y is an offspring of X if
  X is a parent of Y.

This formulation is already close to the formalism of Prolog. The corresponding Prolog clause which has the same meaning is:

```
offspring( Y, X)  :-  parent( X, Y).
```

This clause can also be read as:

For all X and Y,
  if X is a parent of Y then
  Y is an offspring of X.

Prolog clauses such as

```
offspring( Y, X)  :-  parent( X, Y).
```

are called *rules*. There is an important difference between facts and rules. A fact like

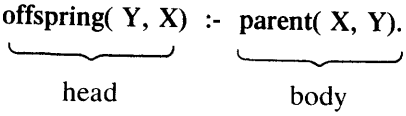
```
parent( tom, liz).
```

is something that is always, unconditionally, true. On the other hand, rules specify things that may be true if some condition is satisfied. Therefore we say that rules have:

- a condition part (the right-hand side of the rule) and

- a conclusion part (the left-hand side of the rule).

The conclusion part is also called the *head* of a clause and the condition part the *body* of a clause. For example:



If the condition **parent(X, Y)** is true then a logical consequence of this is **offspring(Y, X)**.

How rules are actually used by Prolog is illustrated by the following example. Let us ask our program whether Liz is an offspring of Tom:

```
?- offspring( liz, tom).
```

There is no fact about offsprings in the program, therefore the only way to consider this question is to apply the rule about offsprings. The rule is general in the sense that it is applicable to any objects X and Y; therefore it can also be applied to such particular objects as **liz** and **tom**. To apply the rule to **liz** and **tom**, Y has to be substituted with **liz**, and X with **tom**. We say that the variables X and Y become instantiated to:

X = tom and **Y = liz**

After the instantiation we have obtained a special case of our general rule. The special case is:

```
offspring( liz, tom) :- parent( tom, liz).
```

The condition part has become

```
parent( tom, liz)
```

Now Prolog tries to find out whether the condition part is true. So the initial goal

```
offspring( liz, tom)
```

has been replaced with the subgoal

```
parent( tom, liz)
```

This (new) goal happens to be trivial as it can be found as a fact in our program. This means that the conclusion part of the rule is also true, and Prolog will answer the question with **yes**.

Let us now add more family relations to our example program. The

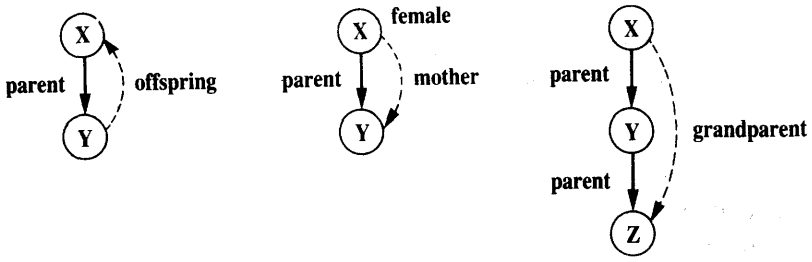


Figure 1.3 Definition graphs for the relations **offspring**, **mother** and **grandparent** in terms of other relations.

specification of the **mother** relation can be based on the following logical statement:

For all X and Y,
X is the mother of Y if
X is a parent of Y and
X is a female.

This is translated into Prolog as the following rule:

```
mother( X, Y)  :-  parent( X, Y), female( X).
```

A comma between two conditions indicates the conjunction of the conditions, meaning that *both* conditions have to be true.

Relations such as **parent, offspring** and **mother** can be illustrated by diagrams such as those in Figure 1.3. These diagrams conform to the following conventions. Nodes in the graphs correspond to objects – that is, arguments of relations. Arcs between nodes correspond to binary (or two-place) relations. The arcs are oriented so as to point from the first argument of the relation to the second argument. Unary relations are indicated in the diagrams by simply marking the corresponding objects with the name of the relation. The relations that are being defined are represented by dashed arcs. So each diagram should be understood as follows: if relations shown by solid arcs hold, then the relation shown by a dashed arc also holds. The **grandparent** relation can be, according to Figure 1.3, immediately written in Prolog as:

```
grandparent( X, Z)  :-  parent( X, Y), parent( Y, Z).
```

At this point it will be useful to make a comment on the layout of our programs. Prolog gives us almost full freedom in choosing the layout of the program. So we can insert spaces and new lines as it best suits our taste. In general we want to make our programs look nice and tidy, and, above all, easy to read. To this end we will often choose to write the head of a clause and each

goal of the body on a separate line. When doing this, we will indent goals in order to make the difference between the head and the goals more visible. For example, the **grandparent** rule would be, according to this convention, written as follows:

```
grandparent( X, Z) :-
  parent( X, Y),
  parent( Y, Z).
```

Figure 1.4 illustrates the **sister** relation:

For any X and Y,
X is a sister of Y if
(1) both X and Y have the same parent, and
(2) X is a female.

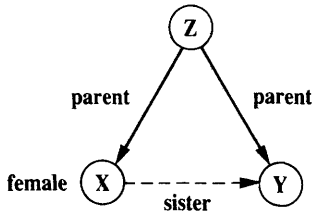


Figure 1.4 Defining the **sister** relation.

The graph in Figure 1.4 can be translated into Prolog as:

```
sister( X, Y) :-
  parent( Z, X),
  parent( Z, Y),
  female( X).
```

Notice the way in which the requirement 'both X and Y have the same parent' has been expressed. The following logical formulation was used: some Z must be a parent of X, and this *same* Z must be a parent of Y. An alternative, but less elegant way would be to say: Z1 is a parent of X, and Z2 is a parent of Y, and Z1 is equal to Z2.

We can now ask:

```
?- sister( ann, pat).
```

The answer will be 'yes', as expected (see Figure 1.1). Therefore we might

conclude that the **sister** relation, as defined, works correctly. There is, however, a rather subtle flaw in our program which is revealed if we ask the question Who is Pat's sister?:

```
?- sister( X, pat).
```

Prolog will find two answers, one of which may come as a surprise:

```
X = ann;
X = pat
```

So, Pat is a sister to herself?! This is probably not what we had in mind when defining the **sister** relation. However, according to our rule about sisters Prolog's answer is perfectly logical. Our rule about sisters does not mention that X and Y must not be the same if X is to be a sister of Y. As this is not required Prolog (rightfully) assumes that X and Y can be the same, and will as a consequence find that any female who has a parent is a sister of herself.

To correct our rule about sisters we have to add that X and Y must be different. We will see in later chapters how this can be done in several ways, but for the moment we will assume that a relation **different** is already known to Prolog, and that

```
different( X, Y)
```

is satisfied if and only if X and Y are not equal. An improved rule for the **sister** relation can then be:

```
sister( X, Y) :-
  parent( Z, X),
  parent( Z, Y),
  female( X),
  different( X, Y).
```

Some important points of this section are:

- Prolog programs can be extended by simply adding new clauses.
- Prolog clauses are of three types: *facts*, *rules* and *questions*.
- *Facts* declare things that are always, unconditionally true.
- *Rules* declare things that are true depending on a given condition.
- By means of *questions* the user can ask the program what things are true.
- Prolog clauses consist of the *head* and the *body*. The body is a list of *goals* separated by commas. Commas are understood as conjunctions.
- Facts are clauses that have the empty body. Questions only have the body. Rules have the head and the (non-empty) body.

- In the course of computation, a variable can be substituted by another object. We say that a variable becomes *instantiated*.
- Variables are assumed to be universally quantified and are read as 'for all'. Alternative readings are, however, possible for variables that appear only in the body. For example

  hasachild(X) :- parent(X, Y).

  can be read in two ways:

  (a) *For all* X and Y,
  if X is a parent of Y then
  X has a child.

  (b) *For all* X,
  X has a child if
  there is *some* Y such that X is a parent of Y.

Exercises

1.3 Translate the following statements into Prolog rules:

(a) Everybody who has a child is happy (introduce a one-argument relation **happy**).

(b) For all X, if X has a child who has a sister then X has two children (introduce new relation **hastwochildren**).

1.4 Define the relation **grandchild** using the **parent** relation. Hint: It will be similar to the **grandparent** relation (see Figure 1.3).

1.5 Define the relation **aunt(X, Y)** in terms of the relations **parent** and **sister**. As an aid you can first draw a diagram in the style of Figure 1.3 for the **aunt** relation.

1.3 A recursive rule definition

Let us add one more relation to our family program, the **predecessor** relation. This relation will be defined in terms of the **parent** relation. The whole definition can be expressed with two rules. The first rule will define the direct (immediate) predecessors and the second rule the indirect predecessors. We say that some X is an indirect predecessor of some Z if there is a parentship chain of people between X and Z, as illustrated in Figure 1.5. In our example of Figure 1.1, Tom is a direct predecessor of Liz and an indirect predecessor of Pat.

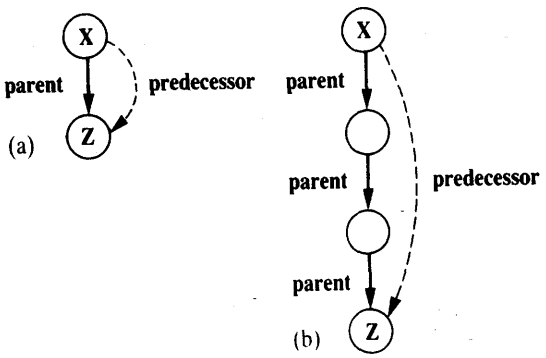


Figure 1.5 Examples of the **predecessor** relation: (a) **X** is a *direct* predecessor of **Z**; (b) **X** is an indirect predecessor of **Z**.

The first rule is simple and can be formulated as:

For all X and Z,
X is a predecessor of Z if
X is a parent of Z.

This is straightforwardly translated into Prolog as:

```
predecessor( X, Z) :-
  parent( X, Z).
```

The second rule, on the other hand, is more complicated because the chain of parents may present some problems. One attempt to define indirect predecessors could be as shown in Figure 1.6. According to this, the **predecessor**

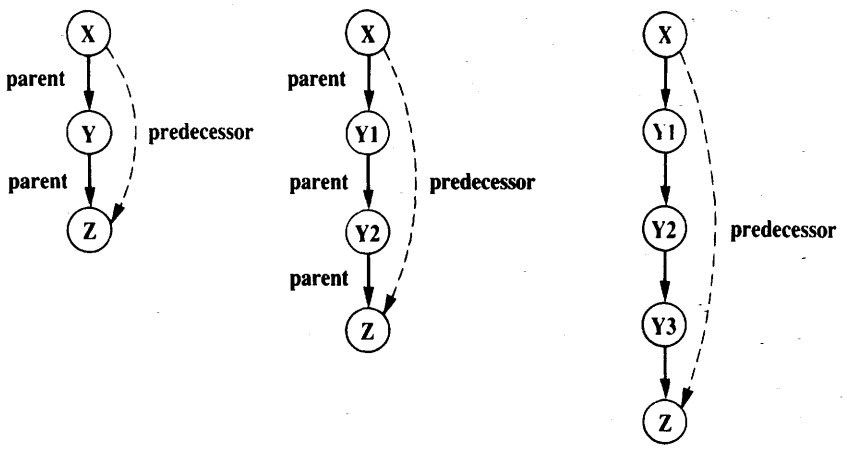


Figure 1.6 Predecessor-successor pairs at various distances.

relation would be defined by a set of clauses as follows:

```
predecessor( X, Z) :-
  parent( X, Z).

predecessor( X, Z) :-
  parent( X, Y),
  parent( Y, Z).

predecessor( X, Z) :-
  parent( X, Y1),
  parent( Y1, Y2),
  parent( Y2, Z).

predecessor( X, Z) :-
  parent( X, Y1),
  parent( Y1, Y2),
  parent( Y2, Y3),
  parent( Y3, Z).

...
```

This program is lengthy and, more importantly, it only works to some extent. It would only discover predecessors to a certain depth in a family tree because the length of the chain of people between the predecessor and the successor would be limited according to the length of our predecessor clauses.

There is, however, an elegant and correct formulation of the **predecessor** relation: it will be correct in the sense that it will work for predecessors at any depth. The key idea is to define the **predecessor** relation in terms of itself. Figure 1.7 illustrates the idea:

For all X and Z,
  X is a predecessor of Z if
  there is a Y such that
  (1) X is a parent of Y and
  (2) Y is a predecessor of Z.

A Prolog clause with the above meaning is:

```
predecessor( X, Z) :-
  parent( X, Y),
  predecessor( Y, Z).
```

We have thus constructed a complete program for the **predecessor** relation, which consists of two rules: one for direct predecessors and one for indirect predecessors. Both rules are rewritten together here:

```
predecessor( X, Z) :-
  parent( X, Z).
```

```
predecessor( X, Z) :-
  parent( X, Y),
  predecessor( Y, Z).
```

The key to this formulation was the use of **predecessor** itself in its definition. Such a definition may look surprising in view of the question: When defining something, can we use this same thing that has not yet been completely defined? Such definitions are, in general, called *recursive* definitions. Logically, they are perfectly correct and understandable, which is also intuitively obvious if we look at Figure 1.7. But will the Prolog system be able to use recursive rules? It turns out that Prolog can indeed very easily use recursive definitions. Recursive programming is, in fact, one of the fundamental principles of programming in Prolog. It is not possible to solve tasks of any significant complexity in Prolog without the use of recursion.

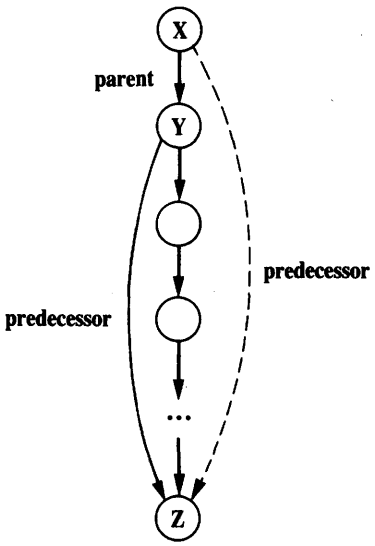


Figure 1.7 Recursive formulation of the **predecessor** relation.

Going back to our program, we can ask Prolog: Who are Pam's successors? That is: Who is a person that Pam is his or her predecessor?

```
?- predecessor( pam, X).

X = bob;

X = ann;

X = pat;

X = jim
```

Prolog's answers are of course correct and they logically follow from our definition of the **predecessor** and the **parent** relation. There is, however, a rather important question: *How* did Prolog actually use the program to find these answers?

An informal explanation of how Prolog does this is given in the next section. But first let us put together all the pieces of our family program, which

```
parent( pam, bob).          % Pam is a parent of Bob
parent( tom, bob).
parent( tom, liz).
parent( bob, ann).
parent( bob, pat).
parent( pat, jim).

female( pam).               % Pam is female
male( tom).                 % Tom is male
male( bob).
female( liz).
female( ann).
female( pat).
male( jim).

offspring( Y, X) :-         % Y is an offspring of X if
  parent( X, Y).            % X is a parent of Y

mother( X, Y) :-            % X is the mother of Y if
  parent( X, Y),            % X is a parent of Y and
  female( X).               % X is female

grandparent( X, Z) :-       % X is a grandparent of Z if
  parent( X, Y),            % X is a parent of Y and
  parent( Y, Z).            % Y is a parent of Z

sister( X, Y) :-            % X is a sister of Y if
  parent( Z, X),
  parent( Z, Y),            % X and Y have the same parent and
  female( X),               % X is female and
  different( X, Y).         % X and Y are different

predecessor( X, Z) :-       % Rule pr1: X is a predecessor of Z
  parent( X, Z).

predecessor( X, Z) :-       % Rule pr2: X is a predecessor of Z
  parent( X, Y),
  predecessor( Y, Z).
```

Figure 1.8 The family program.

was extended gradually by adding new facts and rules. The final form of the program is shown in Figure 1.8. Looking at Figure 1.8, two further points are in order here: the first will introduce the term 'procedure', the second will be about comments in programs.

The program in Figure 1.8 defines several relations – **parent, male, female, predecessor**, etc. The **predecessor** relation, for example, is defined by two clauses. We say that these two clauses are *about* the **predecessor** relation. Sometimes it is convenient to consider the whole set of clauses about the same relation. Such a set of clauses is called a *procedure*.

In Figure 1.8, the two rules about the **predecessor** relation have been distinguished by the names 'pr1' and 'pr2', added as *comments* to the program. These names will be used later as references to these rules. Comments are, in general, ignored by the Prolog system. They only serve as a further clarification to the person who reads the program. Comments are distinguished in Prolog from the rest of the program by being enclosed in special brackets '/*' and '*/'. Thus comments in Prolog look like this:

```
/* This is a comment */
```

Another method, more practical for short comments, uses the percent character '%'. Everything between '%' and the end of the line is interpreted as a comment:

```
% This is also a comment
```

Exercise

1.6 Consider the following alternative definition of the **predecessor** relation:

```
predecessor( X, Z) :-
  parent( X, Z).

predecessor( X, Z) :-
  parent( Y, Z),
  predecessor( X, Y).
```

Does this also seem to be a proper definition of predecessors? Can you modify the diagram of Figure 1.7 so that it would correspond to this new definition?

1.4 How Prolog answers questions

This section gives an informal explanation of *how* Prolog answers questions.

A question to Prolog is always a sequence of one or more goals. To answer a question, Prolog tries to satisfy all the goals. What does it mean to *satisfy* a goal? To satisfy a goal means to demonstrate that the goal is true,

assuming that the relations in the program are true. In other words, to satisfy a goal means to demonstrate that the goal *logically follows* from the facts and rules in the program. If the question contains variables, Prolog also has to find what are the particular objects (in place of variables) for which the goals are satisfied. The particular instantiation of variables to these objects is displayed to the user. If Prolog cannot demonstrate for some instantiation of variables that the goals logically follow from the program, then Prolog's answer to the question will be 'no'.

An appropriate view of the interpretation of a Prolog program in mathematical terms is then as follows: Prolog accepts facts and rules as a set of axioms, and the user's question as a *conjectured theorem*; then it tries to prove this theorem – that is, to demonstrate that it can be logically derived from the axioms.

We will illustrate this view by a classical example. Let the axioms be:

All men are fallible.
Socrates is a man.

A theorem that logically follows from these two axioms is:

Socrates is fallible.

The first axiom above can be rewritten as:

For all X, if X is a man then X is fallible.

Accordingly, the example can be translated into Prolog as follows:

```
fallible( X) :- man( X).        % All men are fallible
man( socrates).                 % Socrates is a man
?- fallible( socrates).         % Socrates is fallible?
yes
```

A more complicated example from the family program of Figure 1.8 is:

```
?- predecessor( tom, pat).
```

We know that **parent(bob, pat)** is a fact. Using this fact and rule *pr1* we can conclude **predecessor(bob, pat)**. This is a *derived* fact: it cannot be found explicitly in our program, but it can be derived from facts and rules in the program. An inference step, such as this, can be written in a more compact form as:

```
parent( bob, pat)  ==>  predecessor( bob, pat)
```

This can be read: from **parent(bob, pat)** it follows **predecessor(bob, pat)**, by

rule *pr1*. Further, we know that **parent(tom, bob)** is a fact. Using this fact and the derived fact **predecessor(bob, pat)** we can conclude **predecessor(tom, pat)**, by rule *pr2*. We have thus shown that our goal statement **predecessor(tom, pat)** is true. This whole inference process of two steps can be written as:

parent(bob, pat) ==> **predecessor(bob, pat)**

parent(tom, bob) *and* **predecessor(bob, pat)** ==> **predecessor(tom, pat)**

We have thus shown *what* can be a sequence of steps that satisfy a goal – that is, make it clear that the goal is true. Let us call this a proof sequence. We have not, however, shown *how* the Prolog system actually finds such a proof sequence.

Prolog finds the proof sequence in the inverse order to that which we have just used. Instead of starting with simple facts given in the program, Prolog starts with the goals and, using rules, substitutes the current goals with new goals, until new goals happen to be simple facts. Given the question

```
?- predecessor( tom, pat).
```

Prolog will try to satisfy this goal. In order to do so it will try to find a clause in the program from which the above goal could immediately follow. Obviously, the only clauses relevant to this end are *pr1* and *pr2*. These are the rules about the **predecessor** relation. We say that the heads of these rules *match* the goal.

The two clauses, *pr1* and *pr2*, represent two alternative ways for Prolog to proceed. Prolog first tries that clause which appears first in the program:

```
predecessor( X, Z) :- parent( X, Z).
```

Since the goal is **predecessor(tom, pat)**, the variables in the rule must be instantiated as follows:

X = tom, Z = pat

The original goal **predecessor(tom, pat)** is then replaced by a new goal:

parent(tom, pat)

This step of using a rule to transform a goal into another goal, as above, is graphically illustrated in Figure 1.9. There is no clause in the program whose head matches the goal **parent(tom, pat)**, therefore this goal fails. Now Prolog *backtracks* to the original goal in order to try an alternative way to derive the top goal **predecessor(tom, pat)**. The rule *pr2* is thus tried:

```
predecessor( X, Z) :-
  parent( X, Y),
  predecessor( Y, Z).
```

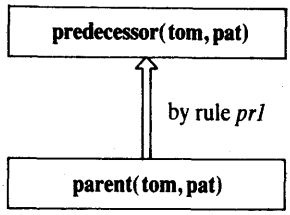


Figure 1.9 The first step of the execution. The top goal is true if the bottom goal is true.

As before, the variables X and Z become instantiated as:

X = tom, Z = pat

But Y is not instantiated yet. The top goal **predecessor(tom, pat)** is replaced by two goals:

parent(tom, Y),
predecessor(Y, pat)

This executional step is shown in Figure 1.10, which is an extension to the situation we had in Figure 1.9.

Being now faced with *two* goals, Prolog tries to satisfy them in the order that they are written. The first one is easy as it matches one of the facts in the program. The matching forces Y to become instantiated to **bob**. Thus the first goal has been satisfied, and the remaining goal has become:

predecessor(bob, pat)

To satisfy this goal the rule *prl* is used again. Note that this (second) application of the same rule has nothing to do with its previous application. Therefore, Prolog uses a new set of variables in the rule each time the rule is applied. To

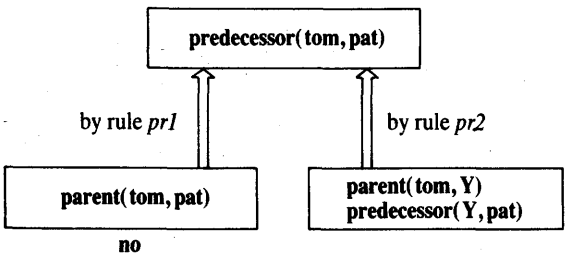


Figure 1.10 Execution trace continued from Figure 1.9.

indicate this we shall rename the variables in rule *pr1* for this application as follows:

```
predecessor( X', Z') :-
  parent( X', Z').
```

The head has to match our current goal **predecessor(bob, pat)**. Therefore

X' = bob, Z' = pat

The current goal is replaced by

parent(bob, pat)

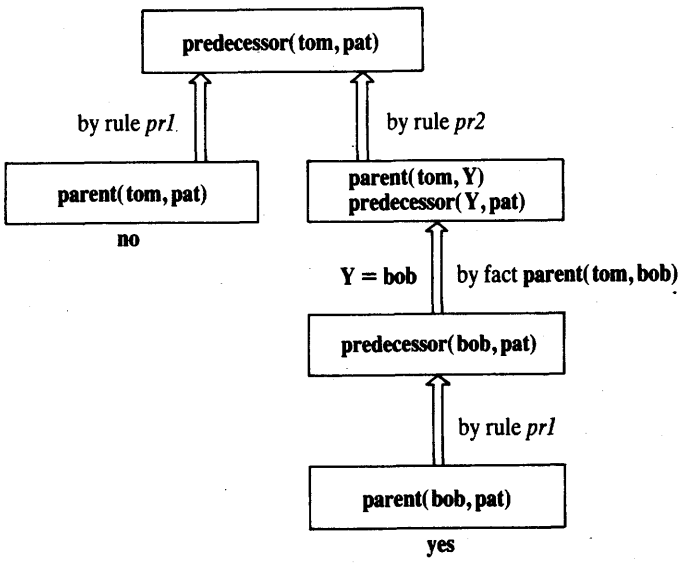


Figure 1.11 The complete execution trace to satisfy the goal **predecessor(tom, pat)**. The right-hand branch proves the goal is satisfiable.

This goal is immediately satisfied because it appears in the program as a fact. This completes the execution trace which is graphically shown in Figure 1.11.

The graphical illustration of the execution trace in Figure 1.11 has the form of a tree. The nodes of the tree correspond to goals, or to lists of goals that are to be satisfied. The arcs between the nodes correspond to the application of (alternative) program clauses that transform the goals at one node into the goals at another node. The top goal is satisfied when a path is found from the root node (top goal) to a leaf node labelled 'yes'. A leaf is labelled 'yes' if it is a simple fact. The execution of Prolog programs is the searching for such paths.

During the search Prolog may enter an unsuccessful branch. When Prolog discovers that a branch fails it automatically *backtracks* to the previous node and tries to apply an alternative clause at that node.

Exercise

1.7 Try to understand how Prolog derives answers to the following questions, using the program of Figure 1.8. Try to draw the corresponding derivation diagrams in the style of Figures 1.9 to 1.11. Will any backtracking occur at particular questions?

(a) ?- **parent(pam, bob).**

(b) ?- **mother(pam, bob).**

(c) ?- **grandparent(pam, ann).**

(d) ?- **grandparent(bob, jim).**

1.5 Declarative and procedural meaning of programs

In our examples so far it has always been possible to understand the results of the program without exactly knowing *how* the system actually found the results. It therefore makes sense to distinguish between two levels of meaning of Prolog programs; namely,

- the *declarative meaning* and
- the *procedural meaning.*

The declarative meaning is concerned only with the *relations* defined by the program. The declarative meaning thus determines *what* will be the output of the program. On the other hand, the procedural meaning also determines *how* this output is obtained; that is, how are the relations actually evaluated by the Prolog system.

The ability of Prolog to work out many procedural details on its own is considered to be one of its specific advantages. It encourages the programmer to consider the declarative meaning of programs relatively independently of their procedural meaning. Since the results of the program are, in principle, determined by its declarative meaning, this should be (in principle) sufficient for writing programs. This is of practical importance because the declarative aspects of programs are usually easier to understand than the procedural details. To take full advantage of this, the programmer should concentrate mainly on the declarative meaning and, whenever possible, avoid being distracted by the executional details. These should be left to the greatest possible extent to the Prolog system itself.

This declarative approach indeed often makes programming in Prolog easier than in typical procedurally oriented programming languages such as Pascal. Unfortunately, however, the declarative approach is not always sufficient. It will later become clear that, especially in large programs, the procedural aspects cannot be completely ignored by the programmer for practical reasons of executional efficiency. Nevertheless, the declarative style of thinking about Prolog programs should be encouraged and the procedural aspects ignored to the extent that is permitted by practical constraints.

Summary

- Prolog programming consists of defining relations and querying about relations.
- A program consists of *clauses*. These are of three types: *facts, rules* and *questions*.
- A relation can be specified by *facts*, simply stating the n-tuples of objects that satisfy the relation, or by stating *rules* about the relation.
- A *procedure* is a set of clauses about the same relation.
- Querying about relations, by means of *questions*, resembles querying a database. Prolog's answer to a question consists of a set of objects that satisfy the question.
- In Prolog, to establish whether an object satisfies a query is often a complicated process that involves logical inference, exploring among alternatives and possibly *backtracking*. All this is done automatically by the Prolog system and is, in principle, hidden from the user.
- Two types of meaning of Prolog programs are distinguished: declarative and procedural. The declarative view is advantageous from the programming point of view. Nevertheless, the procedural details often have to be considered by the programmer as well.
- The following concepts have been introduced in this chapter:

  clause, fact, rule, question
  the head of a clause, the body of a clause
  recursive rule, recursive definition
  procedure
  atom, variable
  instantiation of a variable
  goal
  goal is satisfiable, goal succeeds
  goal is unsatisfiable, goal fails
  backtracking
  declarative meaning, procedural meaning

References

Various implementations of Prolog use different syntactic conventions. In this book we use the so-called Edinburgh syntax (also called DEC-10 syntax, established by the influential implementation of Prolog for the DEC-10 computer; Pereira *et al.* 1978) which has been adopted by many popular Prologs such as Quintus Prolog, CProlog, Poplog, etc.

Bowen, D. L. (1981) *DECsystem-10 Prolog User's Manual.* University of Edinburgh: Department of Artificial Intelligence.

Mellish, C. and Hardy, S. (1984) *Integrating Prolog in the POPLOG environment. Implementations of Prolog* (J. A. Campbell, ed.). Ellis Horwood.

Pereira, F. (1982) *C-Prolog User's Manual.* University of Edinburgh: Department of Computer-Aided Architectural Design.

Pereira, L. M., Pereira, F. and Warren, D. H. D. (1978) *User's Guide to DECsystem-10 Prolog.* University of Edinburgh: Department of Artificial Intelligence.

Quintus Prolog User's Guide and Reference Manual. Palo Alto: Quintus Computer Systems Inc. (1985).

2 Syntax and Meaning of Prolog Programs

This chapter gives a systematic treatment of the syntax and semantics of basic concepts of Prolog, and introduces structured data objects. The topics included are:

- simple data objects (atoms, numbers, variables)
- structured objects
- matching as the fundamental operation on objects
- declarative (or non-procedural) meaning of a program
- procedural meaning of a program
- relation between the declarative and procedural meanings of a program
- altering the procedural meaning by reordering clauses and goals

Most of these topics have already been reviewed in Chapter 1. Here the treatment will become more formal and detailed.

2.1 Data objects

Figure 2.1 shows a classification of data objects in Prolog. The Prolog system recognizes the type of an object in the program by its syntactic form. This is possible because the syntax of Prolog specifies different forms for each type of

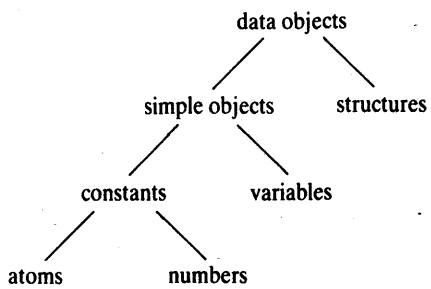


Figure 2.1 Data objects in Prolog.

data objects. We have already seen a method for distinguishing between atoms and variables in Chapter 1: variables start with upper-case letters whereas atoms start with lower-case letters. No additional information (such as data-type declaration) has to be communicated to Prolog in order to recognize the type of an object.

2.1.1 Atoms and numbers

In Chapter 1 we have seen some simple examples of atoms and variables. In general, however, they can take more complicated forms – that is, strings of the following characters:

- upper-case letters A, B, ..., Z
- lower-case letters a, b, ..., z
- digits 0, 1, 2, ..., 9
- special characters such as + − * / < > = : . & _ ~

Atoms can be constructed in three ways:

(1) Strings of letters, digits and the underscore character, '_', starting with a lower-case letter:

anna
nil
x25
x_25
x_25AB
x_
x___y
alpha_beta_procedure
miss_Jones
sarah_jones

(2) Strings of special characters:

<--->
======>
...
.:.
::=

When using atoms of this form, some care is necessary because some strings of special characters already have a predefined meaning; an example is ':-'.

(3) Strings of characters enclosed in single quotes. This is useful if we want, for example, to have an atom that starts with a capital letter. By enclosing

it in quotes we make it distinguishable from variables:

'Tom'
'South_America'
'Sarah Jones'

Numbers used in Prolog include integer numbers and real numbers. The syntax of integers is simple, as illustrated by the following examples:

1 1313 0 −97

Not all integer numbers can be represented in a computer, therefore the range of integers is limited to an interval between some smallest and some largest number permitted by a particular Prolog implementation. Normally the range allowed by an implementation is at least between −16383 and 16383, and often it is considerably wider.

The treatment of real numbers depends on the implementation of Prolog. We will assume the simple syntax of numbers, as shown by the following examples:

3.14 −0.0035 100.2

Real numbers are not used very much in typical Prolog programming. The reason for this is that Prolog is primarily a language for symbolic, non-numeric computation, as opposed to number crunching oriented languages such as Fortran. In symbolic computation, integers are often used, for example, to count the number of items in a list; but there is little need for real numbers.

Apart from this lack of necessity to use real numbers in typical Prolog applications, there is another reason for avoiding real numbers. In general, we want to keep the meaning of programs as neat as possible. The introduction of real numbers somewhat impairs this neatness because of numerical errors that arise due to rounding when doing arithmetic. For example, the evaluation of the expression

10000 + 0.0001 − 10000

may result in 0 instead of the correct result 0.0001.

2.1.2 Variables

Variables are strings of letters, digits and underscore characters. They start with an upper-case letter or an underscore character:

X
Result
Object2
Participant_list

```
ShoppingList
_x23
_23
```

When a variable appears in a clause once only, we do not have to invent a name for it. We can use the so-called 'anonymous' variable, which is written as a single underscore character. For example, let us consider the following rule:

```
hasachild( X) :- parent( X, Y).
```

This rule says: for all X, X has a child if X is a parent of some Y. We are defining the property **hasachild** which, as it is meant here, does not depend on the name of the child. Thus, this is a proper place in which to use an anonymous variable. The clause above can thus be rewritten:

```
hasachild( X) :- parent( X, _).
```

Each time a single underscore character occurs in a clause it represents a new anonymous variable. For example, we can say that there is somebody who has a child if there are two objects such that one is a parent of the other:

```
somebody_has_child :- parent( _, _).
```

This is equivalent to:

```
somebody_has_child :- parent( X, Y).
```

But this is, of course, quite different from:

```
somebody_has_child :- parent( X, X).
```

If the anonymous variable appears in a question clause then its value is not output when Prolog answers the question. If we are interested in people who have children, but not in the names of the children, then we can simply ask:

```
?- parent( X, _).
```

The *lexical scope* of variable names is one clause. This means that, for example, if the name X15 occurs in two clauses, then it signifies two different variables. But each occurrence of X15 within the same clause means the same variable. The situation is different for constants: the same atom always means the same object in any clause – that is, throughout the whole program.

2.1.3 Structures

Structured objects (or simply *structures*) are objects that have several components. The components themselves can, in turn, be structures. For example,

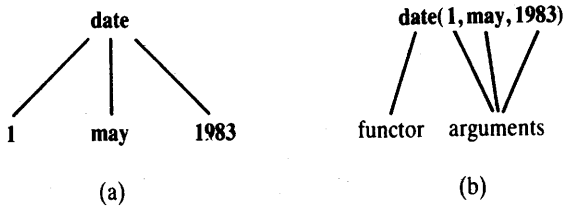


Figure 2.2 Date is an example of a structured object: (a) as it is represented as a tree; (b) as it is written in Prolog.

the date can be viewed as a structure with three components: day, month, year. Although composed of several components, structures are treated in the program as single objects. In order to combine the components into a single object we have to choose a *functor*. A suitable functor for our example is **date**. Then the date 1st May 1983 can be written as:

date(1, may, 1983)

(see Figure 2.2).

All the components in this example are constants (two integers and one atom). Components can also be variables or other structures. Any day in May can be represented by the structure:

date(Day, may, 1983)

Note that **Day** is a variable and can be instantiated to any object at some later point in the execution.

This method for data structuring is simple and powerful. It is one of the reasons why Prolog is so naturally applied to problems that involve symbolic manipulation.

Syntactically, all data objects in Prolog are *terms*. For example,

may

and

date(1, may, 1983)

are terms.

All structured objects can be pictured as trees (see Figure 2.2 for an example). The root of the tree is the functor, and the offsprings of the root are the components. If a component is also a structure then it is a subtree of the tree that corresponds to the whole structured object.

Our next example will show how structures can be used to represent some simple geometric objects (see Figure 2.3). A point in two-dimensional space is

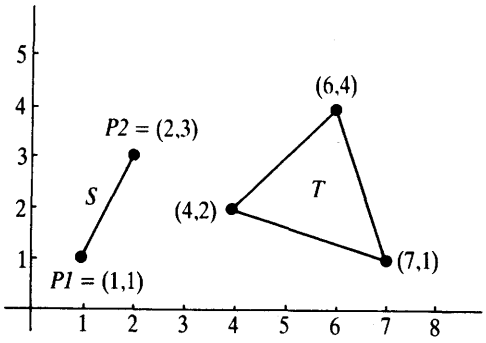


Figure 2.3 Some simple geometric objects.

defined by its two coordinates; a line segment is defined by two points; and a triangle can be defined by three points. Let us choose the following functors:

point	for points,
seg	for line segments, and
triangle	for triangles.

Then the objects in Figure 2.3 can be represented by the following Prolog terms:

```
P1 = point(1,1)
P2 = point(2,3)
S = seg( P1, P2) = seg( point(1,1), point(2,3) )
T = triangle( point(4,2), point(6,4), point(7,1) )
```

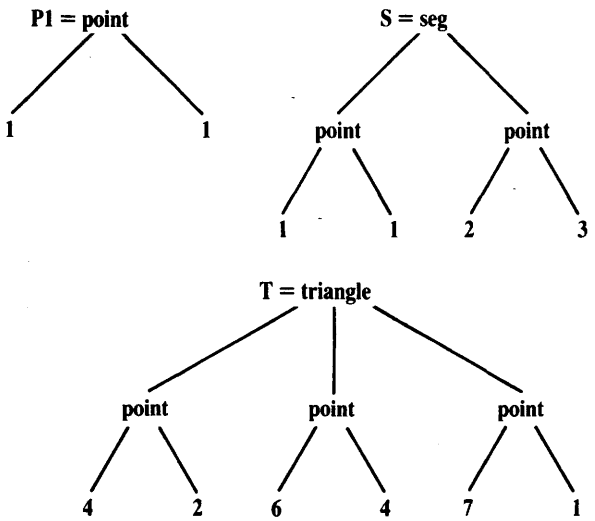


Figure 2.4 Tree representation of the objects in Figure 2.3.

The corresponding tree representation of these objects is shown in Figure 2.4. In general, the functor at the root of the tree is called the *principal functor* of the term.

If in the same program we also had points in three-dimensional space then we could use another functor, **point3**, say, for their representation:

point3(X, Y, Z)

We can, however, use the same name, **point**, for points in both two and three dimensions, and write for example:

point(X1, Y1) and **point(X, Y, Z)**

If the same name appears in the program in two different roles, as is the case for **point** above, the Prolog system will recognize the difference by the number of arguments, and will interpret this name as two functors: one of them with two arguments and the other one with three arguments. This is so because each functor is defined by two things:

(1) the name, whose syntax is that of atoms;
(2) the *arity* – that is, the number of arguments.

As already explained, all structured objects in Prolog are trees, represented in the program by terms. We will study two more examples to illustrate how naturally complicated data objects can be represented by Prolog terms. Figure 2.5 shows the tree structure that corresponds to the arithmetic expression

$(a + b) * (c - 5)$

According to the syntax of terms introduced so far this can be written, using the symbols '*', '+' and '−' as functors, as follows:

***(+(a, b), −(c, 5))**

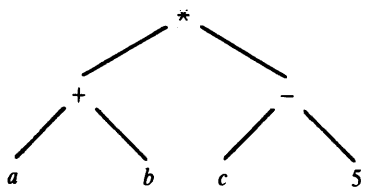


Figure 2.5 A tree structure that corresponds to the arithmetic expression $(a + b)*(c - 5)$.

This is of course a legal Prolog term; but this is not the form that we would normally like to have. We would normally prefer the usual, infix notation as used in mathematics. In fact, Prolog also allows us to use the infix notation so that the symbols '*', '+' and '−' are written as infix operators. Details of how the programmer can define his or her own operators will be discussed in Chapter 3.

As the last example we consider some simple electric circuits shown in Figure 2.6. The right-hand side of the figure shows the tree representation of these circuits. The atoms **r1, r2, r3** and **r4** are the names of the resistors. The

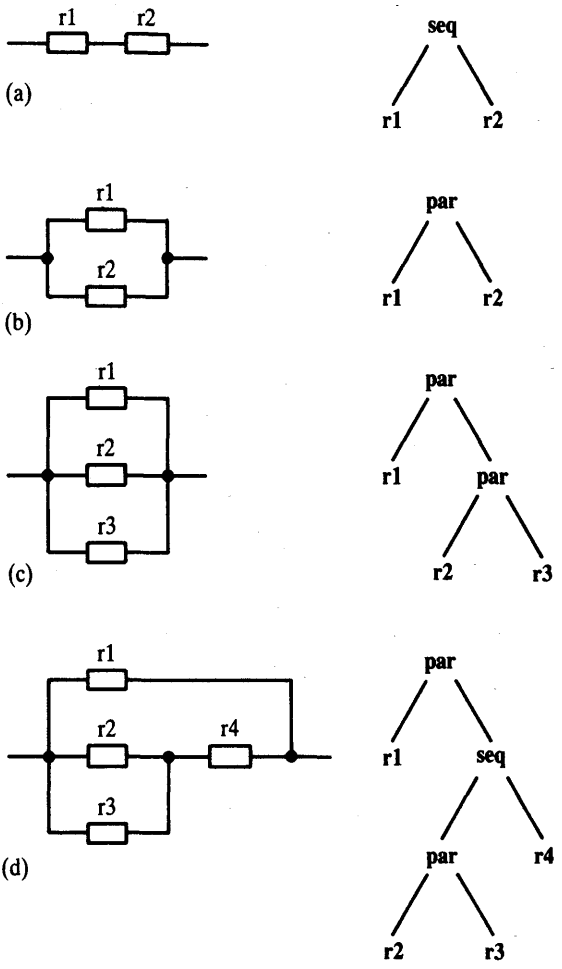


Figure 2.6 Some simple electric circuits and their tree representations: (a) sequential composition of resistors r1 and r2; (b) parallel composition of two resistors; (c) parallel composition of three resistors; (d) parallel composition of r1 and another circuit.

functors **par** and **seq** denote the parallel and the sequential compositions of resistors respectively. The corresponding Prolog terms are:

```
seq( r1, r2)
par( r1, r2)
par( r1, par( r2, r3) )
par( r1, seq( par( r2, r3), r4) )
```

Exercises

2.1 Which of the following are syntactically correct Prolog objects? What kinds of object are they (atom, number, variable, structure)?

(a) **Diana**
(b) **diana**
(c) **'Diana'**
(d) **_diana**
(e) **'Diana goes south'**
(f) **goes(diana, south)**
(g) **45**
(h) **5(X, Y)**
(i) **+(north, west)**
(j) **three(Black(Cats))**

2.2 Suggest a representation for rectangles, squares and circles as structured Prolog objects. Use an approach similar to that in Figure 2.4. For example, a rectangle can be represented by four points (or maybe three points only). Write some example terms that represent some concrete objects of these types using the suggested representation.

2.2 Matching

In the previous section we have seen how terms can be used to represent complex data objects. The most important operation on terms is *matching*. Matching alone can produce some interesting computation.

Given two terms, we say that they *match* if:

(1) they are identical, or
(2) the variables in both terms can be instantiated to objects in such a way that after the substitution of variables by these objects the terms become identical.

For example, the terms **date(D, M, 1983)** and **date(D1, may, Y1)** match. One instantiation that makes both terms identical is:

- **D** is instantiated to **D1**
- **M** is instantiated to **may**
- **Y1** is instantiated to **1983**

This instantiation is more compactly written in the familiar form in which Prolog outputs results:

```
D = D1
M = may
Y1 = 1983
```

On the other hand, the terms **date(D, M, 1983)** and **date(D1, M1, 1444)** do not match, nor do the terms **date(X, Y, Z)** and **point(X, Y, Z)**.

Matching is a process that takes as input two terms and checks whether they match. If the terms do not match we say that this process *fails*. If they do match then the process *succeeds* and it also instantiates the variables in both terms to such values that the terms become identical.

Let us consider again the matching of the two dates. The request for this operation can be communicated to the Prolog system by the following question, using the operator '=':

```
?- date( D, M, 1983) = date( D1, may, Y1).
```

We have already mentioned the instantiation **D = D1, M = may, Y1 = 1983**, which achieves the match. There are, however, other instantiations that also make both terms identical. Two of them are as follows:

```
D = 1
D1 = 1
M = may
Y1 = 1983

D = third
D1 = third
M = may
Y1 = 1983
```

These two instantiations are said to be *less general* than the first one because they constrain the values of the variables D and D1 stronger than necessary. For making both terms in our example identical, it is only important that D and D1 have the same value, although this value can be anything. Matching in Prolog always results in the *most general* instantiation. This is the instantiation that commits the variables to the least possible extent, thus leaving the greatest

possible freedom for further instantiations if further matching is required. As an example consider the following question:

```
?- date( D, M, 1983)  =  date( D1, may, Y1),
   date( D, M, 1983)  =  date( 15, M, Y).
```

To satisfy the first goal, Prolog instantiates the variables as follows:

```
D = D1
M = may
Y1 = 1983
```

After having satisfied the second goal, the instantiation becomes more specific as follows:

```
D = 15
D1 = 15
M = may
Y1 = 1983
Y = 1983
```

This example also illustrates that variables, during the execution of consecutive goals, typically become instantiated to increasingly more specific values.

The general rules to decide whether two terms, S and T, match are as follows:

(1) If S and T are constants then S and T match only if they are the same object.

(2) If S is a variable and T is anything, then they match, and S is instantiated to T. Conversely, if T is a variable then T is instantiated to S.

(3) If S and T are structures then they match only if
   (a) S and T have the same principal functor, and
   (b) all their corresponding components match.

   The resulting instantiation is determined by the matching of the components.

The last of these rules can be visualized by considering the tree representation of terms, as in the example of Figure 2.7. The matching process starts at the root (the principal functors). As both functors match, the process proceeds to the arguments where matching of the pairs of corresponding arguments occurs. So the whole matching process can be thought of as consisting of the

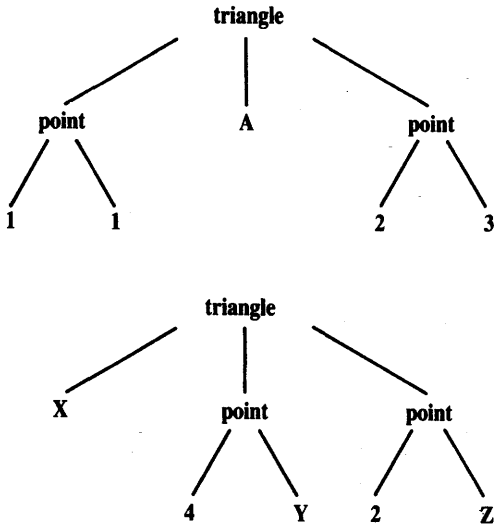


Figure 2.7 Matching **triangle(point(1,1), A, point(2,3)) = triangle(X, point(4,Y), point(2,Z)).**

following sequence of (simpler) matching operations:

triangle = triangle,
point(1,1) = X,
A = point(4,Y),
point(2,3) = point(2,Z).

The whole matching process succeeds because all the matchings in the sequence succeed. The resulting instantiation is:

X = point(1,1)
A = point(4,Y)
Z = 3

The following example will illustrate how matching alone can be used for interesting computation. Let us return to the simple geometric objects of Figure 2.4, and define a piece of program for recognizing horizontal and vertical line segments. 'Vertical' is a property of segments, so it can be formalized in Prolog as a unary relation. Figure 2.8 helps to formulate this relation. A segment is vertical if the x-coordinates of its end-points are equal, otherwise there is no other restriction on the segment. The property 'horizontal' is similarly formulated, with only x and y interchanged. The following program, consisting of two facts, does the job:

vertical(seg(point(X,Y), point(X,Y1)).

horizontal(seg(point(X,Y), point(X1,Y)).

The following conversation is possible with this program:

```
?- vertical( seg( point(1,1), point(1,2) ) ).
yes
?- vertical( seg( point(1,1), point(2,Y) ) ).
no
?- horizontal( seg( point(1,1), point(2,Y) ) ).
Y = 1
```

The first question was answered 'yes' because the goal in the question matched one of the facts in the program. For the second question no match was possible. In the third question, Y was forced to become 1 by matching the fact about horizontal segments.

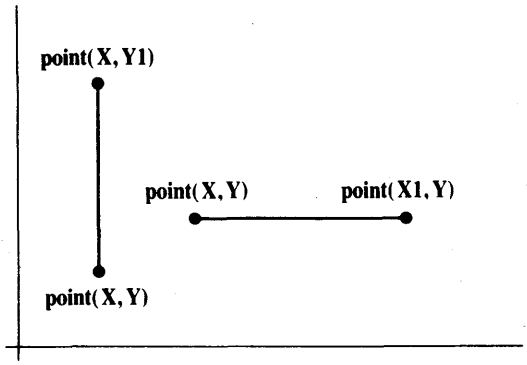


Figure 2.8 Illustration of vertical and horizontal line segments.

A more general question to the program is: Are there any vertical segments that start at the point (2,3)?

```
?- vertical( seg( point(2,3), P) ).
P = point(2,Y)
```

This answer means: Yes, any segment that ends at any point (2,Y), which means anywhere on the vertical line $x = 2$. It should be noted that Prolog's actual answer would probably not look as neat as above, but (depending on the Prolog implementation used) something like this:

```
P = point(2,_136)
```

This is, however, only a cosmetic difference. Here _136 is a variable that has

not been instantiated. _136 is, of course, a legal variable name that the system has constructed during the execution. The system has to generate new names in order to rename the user's variables in the program. This is necessary for two reasons: first, because the same name in different clauses signifies different variables, and second, in successive applications of the same clause, its 'copy' with a new set of variables is used each time.

Another interesting question to our program is: Is there a segment that is both vertical and horizontal?

```
?- vertical( S), horizontal( S).

S = seg( point(X,Y), point(X,Y) )
```

This answer by Prolog says: Yes, any segment that is degenerated to a point has the property of being vertical and horizontal at the same time. The answer was, again, derived simply by matching. As before, some internally generated names may appear in the answer, instead of the variable names X and Y.

Exercises

2.3 Will the following matching operations succeed or fail? If they succeed, what are the resulting instantiations of variables?

(a) **point(A, B) = point(1, 2)**

(b) **point(A, B) = point(X, Y, Z)**

(c) **plus(2, 2) = 4**

(d) **+(2, D) = +(E, 2)**

(e) **triangle(point(-1,0), P2, P3) = triangle(P1, point(1,0), point(0,Y))**

The resulting instantiation defines a family of triangles. How would you describe this family?

2.4 Using the representation for line segments as described in this section, write a term that represents any vertical line segment at $x = 5$.

2.5 Assume that a rectangle is represented by the term **rectangle(P1, P2, P3, P4)** where the P's are the vertices of the rectangle positively ordered. Define the relation

regular(R)

which is true if R is a rectangle whose sides are vertical and horizontal.

2.3 Declarative meaning of Prolog programs

We have already seen in Chapter 1 that Prolog programs can be understood in two ways: declaratively and procedurally. In this and the next section we will

consider a more formal definition of the declarative and procedural meanings of programs in basic Prolog. But first let us look at the difference between these two meanings again.

Consider a clause

```
P :- Q, R.
```

where P, Q and R have the syntax of terms. Some alternative declarative readings of this clause are:

P is true if Q and R are true.

From Q and R follows P.

Two alternative procedural readings of this clause are:

To solve problem P, *first* solve the subproblem Q and *then* the subproblem R.

To satisfy P, *first* satisfy Q and *then* R.

Thus the difference between the declarative readings and the procedural ones is that the latter do not only define the logical relations between the head of the clause and the goals in the body, but also the *order* in which the goals are processed.

Let us now formalize the declarative meaning.

The declarative meaning of programs determines whether a given goal is true, and if so, for what values of variables it is true. To precisely define the declarative meaning we need to introduce the concept of *instance* of a clause. An instance of a clause C is the clause C with each of its variables substituted by some term. A *variant* of a clause C is such an instance of the clause C where each variable is substituted by another variable. For example, consider the clause:

```
hasachild( X) :- parent( X, Y).
```

Two variants of this clause are:

```
hasachild( A) :- parent( A, B).
hasachild( X1) :- parent( X1, X2).
```

Instances of this clause are:

```
hasachild( peter) :- parent( peter, Z).
hasachild( barry) :- parent( barry, small(caroline) ).
```

Given a program and a goal G, the declarative meaning says:

A goal G is true (that is, satisfiable, or logically follows from the program) if and only if

(1) there is a clause C in the program such that
(2) there is a clause instance I of C such that
(a) the head of I is identical to G, and
(b) all the goals in the body of I are true.

This definition extends to Prolog questions as follows. In general, a question to the Prolog system is a *list* of goals separated by commas. A list of goals is true if *all* the goals in the list are true for the *same* instantiation of variables. The values of the variables result from the most general instantiation.

A comma between goals thus denotes the *conjunction* of goals: they *all* have to be true. But Prolog also accepts the *disjunction* of goals: *any one* of the goals in a disjunction has to be true. Disjunction is indicated by a semicolon. For example,

```
P :- Q; R.
```

is read: P is true if Q is true *or* R is true. The meaning of this clause is thus the same as the meaning of the following two clauses together:

```
P :- Q.
P :- R.
```

The comma binds stronger than the semicolon. So the clause

```
P :- Q, R; S, T, U.
```

is understood as

```
P :- ( Q, R); ( S, T, U).
```

and means the same as the clauses:

```
P :- Q, R.
P :- S, T, U.
```

Exercises

2.6 Consider the following program:

```
f( 1, one).
f( s(1), two).
```

```
f( s(s(1)), three).
f( s(s(s(X))), N) :-
  f( X, N).
```

How will Prolog answer the following questions? Whenever several answers are possible, give at least two.

(a) ?- f(s(1), A).

(b) ?- f(s(s(1)), two).

(c) ?- f(s(s(s(s(s(s(1)))))), C).

(d) ?- f(D, three).

2.7 The following program says that two people are relatives if
(a) one is a predecessor of the other, or
(b) they have a common predecessor, or
(c) they have a common successor:

```
relatives( X, Y) :-
  predecessor( X, Y).

relatives( X, Y) :-
  predecessor( Y, X).

relatives( X, Y) :-          % X and Y have a common predecessor
  predecessor( Z, X),
  predecessor( Z, Y).

relatives( X, Y) :-          % X and Y have a common successor
  predecessor( X, Z),
  predecessor( Y, Z).
```

Can you shorten this program by using the semicolon notation?

2.8 Rewrite the following program without using the semicolon notation.

```
translate( Number, Word) :-
  Number = 1, Word = one;
  Number = 2, Word = two;
  Number = 3, Word = three.
```

2.4 Procedural meaning

The procedural meaning specifies *how* Prolog answers questions. To answer a question means to try to satisfy a list of goals. They can be satisfied if the variables that occur in the goals can be instantiated in such a way that the goals logically follow from the program. Thus the procedural meaning of Prolog is a procedure for executing a list of goals with respect to a given program. To 'execute goals' means: try to satisfy them.

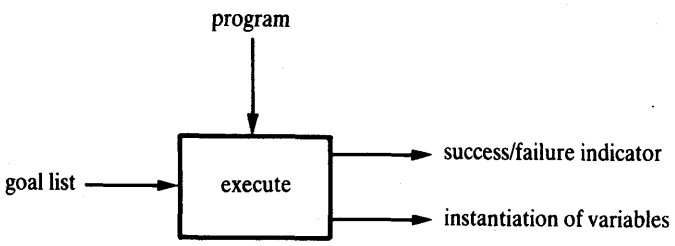


Figure 2.9 Input/output view of the procedure that executes a list of goals.

Let us call this procedure **execute**. As shown in Figure 2.9, the inputs to and the outputs from this procedure are:

input: a program and a goal list

output: a success/failure indicator and an instantiation of variables

The meaning of the two output results is as follows:

(1) The success/failure indicator is 'yes' if the goals are satisfiable and 'no' otherwise. We say that 'yes' signals a *successful* termination and 'no' a *failure*.

(2) An instantiation of variables is only produced in the case of a successful termination; in the case of failure there is no instantiation.

In Chapter 1, we have in effect already discussed informally what procedure **execute** does, under the heading 'How Prolog answers questions?'. What follows in the rest of this section is just a more formal and systematic description of this process, and can be skipped without seriously affecting the understanding of the rest of the book.

Particular operations in the goal execution process are illustrated by the example in Figure 2.10. It may be helpful to study Figure 2.10 before reading the following general description.

PROGRAM

```
big( bear).                  % Clause 1
big( elephant).              % Clause 2
small( cat).                 % Clause 3

brown( bear).                % Clause 4
black( cat).                 % Clause 5
gray( elephant).             % Clause 6

dark( Z) :-                  % Clause 7: Anything black is dark
  black( Z).

dark( Z) :-                  % Clause 8: Anything brown is dark
  brown( Z).
```

QUESTION

```
?- dark( X), big( X).        % Who is dark and big?
```

EXECUTION TRACE

(1) Initial goal list: **dark(X), big(X).**

(2) Scan the program from top to bottom looking for a clause whose head matches the first goal **dark(X)**. Clause 7 found:

```
dark( Z) :- black( Z).
```

Replace the first goal by the instantiated body of clause 7, giving a new goal list.

```
black( X), big( X)
```

(3) Scan the program to find a match with **black(X)**. Clause 5 found: **black(cat)**. This clause has no body, so the goal list, properly instantiated, shrinks to:

```
big( cat)
```

(4) Scan the program for the goal **big(cat)**. No clause found. Therefore backtrack to step (3) and undo the instantiation **X = cat**. Now the goal list is again:

```
black( X), big( X)
```

Continue scanning the program below clause 5. No clause found. Therefore backtrack to step (2) and continue scanning below clause 7. Clause 8 is found:

```
dark( Z) :- brown( Z).
```

Replace the first goal in the goal list by **brown(X)**, giving:

```
brown( X), big( X)
```

(5) Scan the program to match **brown(X)**, finding **brown(bear)**. This clause has no body, so the goal list shrinks to:

```
big( bear)
```

(6) Scan the program and find clause **big(bear)**. It has no body so the goal list shrinks to empty. This indicates successful termination, and the corresponding variable instantiation is:

```
X = bear
```

Figure 2.10 An example to illustrate the procedural meaning of Prolog: a sample trace of the procedure **execute**.

To execute a list of goals

G1, G2, ..., Gm

the procedure **execute** does the following:

- If the goal list is empty then terminate with *success*.
- If the goal list is not empty then continue with (the following) operation called 'SCANNING'.
- *SCANNING*: Scan through the clauses in the program from top to bottom until the first clause, C, is found such that the head of C matches the first goal G1. If there is no such clause then terminate with *failure*.

  If there is such a clause C of the form

  H :- B1, ..., Bn.

  then rename the variables in C to obtain a variant C′ of C, such that C′ and the list G1, ..., Gm have no common variables. Let C′ be

  H′ :- B1′, ..., Bn′.

  Match G1 and H′; let the resulting instantiation of variables be S.

  In the goal list G1, G2, ..., Gm, replace G1 with the list B1′, ..., Bn′, obtaining a new goal list

  B1′, ..., Bn′, G2, ..., Gm

  (Note that if C is a fact then $n = 0$ and the new goal list is shorter than the original one; such shrinking of the goal list may eventually lead to the empty list and thereby a successful termination.)

  Substitute the variables in this new goal list with new values as specified in the instantiation S, obtaining another goal list

  B1″, ..., Bn″, G2′, ..., Gm′

- Execute (recursively with this same procedure) this new goal list. If the execution of this new goal list terminates with success then terminate the execution of the original goal list also with success. If the execution of the new goal list is not successful then abandon this new goal list and go back to SCANNING through the program. Continue the scanning with the clause that immediately follows the clause C (C is the clause that was last used) and try to find a successful termination using some other clause.

This procedure is more compactly written in a Pascal-like notation in Figure 2.11.

Several additional remarks are in order here regarding the procedure **execute** as presented. First, it was not explicitly described how the final resulting instantiation of variables is produced. It is the instantiation S which led to a successful termination, and was possibly further refined by additional instantiations that were done in the nested recursive calls to **execute**.

Whenever a recursive call to **execute** fails, the execution returns to SCANNING, continuing at the program clause C that had been last used before. As the application of the clause C did not lead to a successful termination Prolog has to try an alternative clause to proceed. What effectively happens is that Prolog abandons this whole part of the unsuccessful execution and backtracks to the point (clause C) where this failed branch of the execution was started. When the procedure backtracks to a certain point, all the variable instantiations that were done after that point are undone. This ensures that Prolog systematically examines all the possible alternative paths of execution until one is found that eventually succeeds, or until all of them have been shown to fail.

We have already seen that even after a successful termination the user can force the system to backtrack to search for more solutions. In our description of **execute** this detail was left out.

Of course, in actual implementations of Prolog, several other refinements have to be added to **execute**. One of them is to reduce the amount of

---

procedure *execute* (*Program*, *GoalList*, *Success*);

Input arguments:
  Program: list of clauses
  GoalList: list of goals
Output argument:
  Success: truth value; *Success* will become true if
    GoalList is true with respect to *Program*
Local variables:
  Goal: goal
  OtherGoals: list of goals
  Satisfied: truth value
  MatchOK: truth value
  Instant: instantiation of variables
  H, *H'*, *B1*, *B1'*, ..., *Bn*, *Bn'*: goals
Auxiliary functions:
  empty(L): returns true if L is the empty list
  head(L): returns the first element of list L
  tail(L): returns the rest of L
  append(L1,L2): appends list L2 at the end of list L1
  match(T1,T2,MatchOK,Instant): tries to match terms T1 and T2; if
    succeeds then *MatchOK* is true and *Instant* is the corresponding
    instantiation of variables
  substitute(Instant,Goals): substitutes variables in *Goals* according
    to instantiation *Instant*

```
begin
  if empty(GoalList) then Success := true
  else
    begin
      Goal := head(GoalList);
      OtherGoals := tail(GoalList);
      Satisfied := false;
      while not Satisfied and "more clauses in program" do
        begin
          Let next clause in Program be
            H :- B1, ..., Bn.
          Construct a variant of this clause
            H' :- B1', ..., Bn'.
          match(Goal,H',MatchOK,Instant);
          if MatchOK then
            begin
              NewGoals := append([B1',...,Bn'],OtherGoals);
              NewGoals := substitute(Instant,NewGoals);
              execute(Program,NewGoals,Satisfied)
            end
        end;
      Success := Satisfied
    end
end;
```

Figure 2.11 Executing Prolog goals.

scanning through the program clauses to improve efficiency. So a practical Prolog implementation will not scan through all the clauses of the program, but will only consider the clauses about the relation in the current goal.

Exercise

2.9 Consider the program in Figure 2.10 and simulate, in the style of Figure 2.10, Prolog's execution of the question:

```
?- big( X), dark( X).
```

Compare your execution trace with that of Figure 2.10 when the question was essentially the same, but with the goals in the order:

```
?- dark( X), big( X).
```

In which of the two cases does Prolog have to do more work before the answer is found?

2.5 Example: monkey and banana

The monkey and banana problem is often used as a simple example of problem solving. Our Prolog program for this problem will show how the mechanisms of matching and backtracking can be used in such exercises. We will develop the program in the non-procedural way, and then study its procedural behaviour in detail. The program will be compact and illustrative.

We will use the following variation of the problem. There is a monkey at the door into a room. In the middle of the room a banana is hanging from the ceiling. The monkey is hungry and wants to get the banana, but he cannot stretch high enough from the floor. At the window of the room there is a box the monkey may use. The monkey can perform the following actions: walk on the floor, climb the box, push the box around (if it is already at the box) and grasp the banana if standing on the box directly under the banana. Can the monkey get the banana?

One important task in programming is that of finding a representation of the problem in terms of concepts of the programming language used. In our case we can think of the 'monkey world' as always being in some *state* that can change in time. The current state is determined by the positions of the objects. For example, the initial state of the world is determined by:

(1) Monkey is at door.
(2) Monkey is on floor.
(3) Box is at window.
(4) Monkey does not have banana.

It is convenient to combine all of these four pieces of information into one structured object. Let us choose the word 'state' as the functor to hold the four components together. Figure 2.12 shows the initial state represented as a structured object.

Our problem can be viewed as a one-person game. Let us now formalize the rules of the game. First, the goal of the game is a situation in which the monkey has the banana; that is, any state in which the last component is 'has':

```
state( _, _, _, has)
```

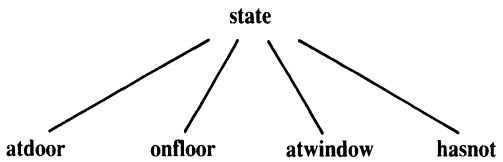


Figure 2.12 The initial state of the monkey world represented as a structured object. The four components are: horizontal position of monkey, vertical position of monkey, position of box, monkey has or has not the banana.

Second, what are the allowed moves that change the world from one state to another? There are four types of moves:

(1) grasp banana,
(2) climb box,
(3) push box,
(4) walk around.

Not all moves are possible in every possible state of the world. For example, the move 'grasp' is only possible if the monkey is standing on the box directly under the banana (which is in the middle of the room) and does not have the banana yet. Such rules can be formalized in Prolog as a three-place relation named **move**:

```
move( State1, M, State2)
```

The three arguments of the relation specify a move thus:

```
State1 --------> State2
          M
```

State1 is the state before the move, M is the move executed and **State2** is the state after the move.

The move 'grasp', with its necessary precondition on the state before the move, can be defined by the clause:

```
move( state( middle, onbox, middle, hasnot),     % Before move
      grasp,                                     % Move
      state( middle, onbox, middle, has) ).      % After move
```

This fact says that after the move the monkey has the banana, and he has remained on the box in the middle of the room.

In a similar way we can express the fact that the monkey on the floor can walk from any horizontal position P1 to any position P2. The monkey can do this regardless of the position of the box and whether it has the banana or not. All this can be defined by the following Prolog fact:

```
move( state( P1, onfloor, B, H),
      walk( P1, P2),                             % Walk from P1 to P2
      state( P2, onfloor, B, H) ).
```

Note that this clause says many things, including, for example:

- the move executed was 'walk from some position P1 to some position P2';
- the monkey is on the floor before and after the move;

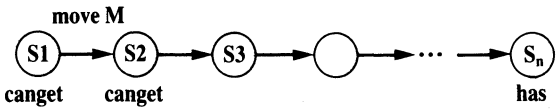


Figure 2.13 Recursive formulation of **canget**.

- the box is at some point B which remained the same after the move;
- the 'has banana' status remains the same after the move.

The clause actually specifies a whole set of possible moves because it is applicable to any situation that matches the specified state before the move. Such a specification is therefore sometimes also called a move *schema*. Due to the concept of Prolog variables such schemas can be easily programmed in Prolog.

The other two types of moves, 'push' and 'climb', can be similarly specified.

The main kind of question that our program will have to answer is: Can the monkey in some initial state S get the banana? This can be formulated as a predicate

canget(S)

where the argument S is a state of the monkey world. The program for **canget** can be based on two observations:

(1) For any state S in which the monkey already has the banana, the predicate **canget** must certainly be true; no move is needed in this case. This corresponds to the Prolog fact:

```
canget( state( _, _, _, has) ).
```

(2) In other cases one or more moves are necessary. The monkey can get the banana in any state S1 if there is some move M from state S1 to some state S2, such that the monkey can then get the banana in state S2 (in zero or more moves). This principle is illustrated in Figure 2.13. A Prolog clause that corresponds to this rule is:

```
canget( S1) :-
  move( S1, M, S2),
  canget( S2).
```

This completes our program which is shown in Figure 2.14.

The formulation of **canget** is recursive and is similar to that of the **predecessor** relation of Chapter 1 (compare Figures 2.13 and 1.7). This principle is used in Prolog again and again.

```
% Legal moves

move( state( middle, onbox, middle, hasnot),
      grasp,                                    % Grasp banana
      state( middle, onbox, middle, has) ).

move( state( P, onfloor, P, H),
      climb,                                    % Climb box
      state( P, onbox, P, H) ).

move( state( P1, onfloor, P1, H),
      push( P1, P2),                            % Push box from P1 to P2
      state( P2, onfloor, P2, H) ).

move( state( P1, onfloor, B, H),
      walk( P1, P2),                            % Walk from P1 to P2
      state( P2, onfloor, B, H) ).

% canget( State): monkey can get banana in State

canget( state( _, _, _, has) ).                 % can 1: Monkey already has it

canget( State1) :-                              % can 2: Do some work to get it
  move( State1, Move, State2),                  % Do something
  canget( State2).                              % Get it now
```

Figure 2.14 A program for the monkey and banana problem.

We have developed our monkey and banana program in the non-procedural way. Let us now study its *procedural* behaviour by considering the following question to the program:

```
?- canget( state( atdoor, onfloor, atwindow, hasnot) ).
```

Prolog's answer is 'yes'. The process carried out by Prolog to reach this answer proceeds, according to the procedural semantics of Prolog, through a sequence of goal lists. It involves some search for right moves among the possible alternative moves. At some point this search will take a wrong move leading to a dead branch. At this stage, backtracking will help it to recover. Figure 2.15 illustrates this search process.

To answer the question Prolog had to backtrack once only. A right sequence of moves was found almost straight away. The reason for this efficiency of the program was the order in which the clauses about the **move** relation occurred in the program. The order in our case (luckily) turned out to be quite suitable. However, less lucky orderings are possible. According to the rules of the game, the monkey could just as easily try to walk here or there

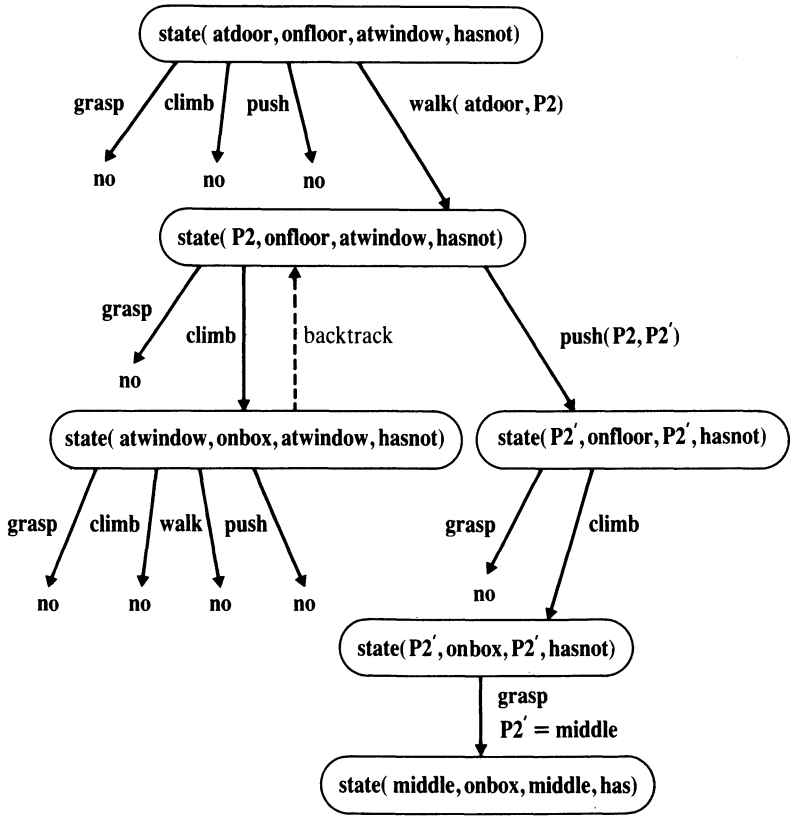


Figure 2.15 The monkey's search for the banana. The search starts at the top node and proceeds downwards, as indicated. Alternative moves are tried in the left-to-right order. Backtracking occurred once only.

without ever touching the box, or aimlessly push the box around. A more thorough investigation will reveal, as shown in the following section, that the ordering of clauses is, in the case of our program, in fact critical.

2.6 Order of clauses and goals

2.6.1 Danger of indefinite looping

Consider the following clause:

```
p :- p.
```

This says that 'p is true if p is true'. This is declaratively perfectly correct, but

procedurally is quite useless. In fact, such a clause can cause problems to Prolog. Consider the question:

```
?- p.
```

Using the clause above, the goal p is replaced by the same goal p; this will be in turn replaced by p, etc. In such a case Prolog will enter an infinite loop not noticing that no progress is being made.

This example is a simple way of getting Prolog to loop indefinitely. However, similar looping could have occurred in some of our previous example programs if we changed the order of clauses, or the order of goals in the clauses. It will be instructive to consider some examples.

In the monkey and banana program, the clauses about the **move** relation were ordered thus: grasp, climb, push, walk (perhaps 'unclimb' should be added for completeness). These clauses say that grasping is possible, climbing is possible, etc. According to the procedural semantics of Prolog, the order of clauses indicates that the monkey prefers grasping to climbing, climbing to pushing, etc. This order of preferences in fact helps the monkey to solve the problem. But what could happen if the order was different? Let us assume that the 'walk' clause appears first. The execution of our original goal of the previous section

```
?- canget( state( atdoor, onfloor, atwindow, hasnot) ).
```

would this time produce the following trace. The first four goal lists (with variables appropriately renamed) are the same as before:

(1) **canget(state(atdoor, onfloor, atwindow, hasnot))**

The second clause of **canget** ('can2') is applied, producing:

(2) **move(state(atdoor, onfloor, atwindow, hasnot), M′, S2′),**
canget(S2′)

By the move **walk(atdoor, P2′)** we get:

(3) **canget(state(P2′, onfloor, atwindow, hasnot))**

Using the clause 'can2' again the goal list becomes:

(4) **move(state(P2′, onfloor, atwindow, hasnot), M″, S2″),**
canget(S2″)

Now the difference occurs. The first clause whose head matches the first goal above is now 'walk' (and not 'climb' as before). The instantiation is

S2″ = state(P2″, onfloor, atwindow, hasnot). Therefore the goal list becomes:

(5) **canget(state(P2″, onfloor, atwindow, hasnot))**

Applying the clause 'can2' we obtain:

(6) **move(state(P2″, onfloor, atwindow, hasnot), M‴, S2‴),**
canget(S2‴)

Again, 'walk' is now tried first, producing:

(7) **canget(state(P2‴, onfloor, atwindow, hasnot))**

Let us now compare the goals (3), (5) and (7). They are the same apart from one variable; this variable is, in turn, P′, P″ and P‴. As we know, the success of a goal does not depend on particular names of variables in the goal. This means that from goal list (3) the execution trace shows no progress. We can see, in fact, that the same two clauses, 'can2' and 'walk', are used repetitively. The monkey walks around without ever trying to use the box. As there is no progress made this will (theoretically) go on for ever: Prolog will not realize that there is no point in continuing along this line.

This example shows Prolog trying to solve a problem in such a way that a solution is never reached, although a solution exists. Such situations are not unusual in Prolog programming. Infinite loops are, also, not unusual in other programming languages. What *is* unusual in comparison with other languages is that the declarative meaning of a Prolog program may be correct, but the program is at the same time procedurally incorrect in that it is not able to produce an answer to a question. In such cases Prolog may not be able to satisfy a goal because it tries to reach an answer by choosing a wrong path.

A natural question to ask at this point is: Can we not make some more substantial change to our program so as to drastically prevent any danger of looping? Or shall we always have to rely just on a suitable ordering of clauses and goals? As it turns out programs, especially large ones, would be too fragile if they just had to rely on some suitable ordering. There are several other methods that preclude infinite loops, and these are much more general and robust than the ordering method itself. These techniques will be used regularly later in the book, especially in those chapters that deal with path finding, problem solving and search.

2.6.2 Program variations through reordering of clauses and goals

Already in the example programs of Chapter 1 there was a latent danger of producing a cycling behaviour. Our program to specify the **predecessor** relation

in Chapter 1 was:

```
predecessor( X, Z) :-
  parent( X, Z).

predecessor( X, Z) :-
  parent( X, Y),
  predecessor( Y, Z).
```

Let us analyze some variations of this program. All the variations will clearly have the same declarative meaning, but not the same procedural meaning.

```
% Four versions of the predecessor program

% The original version

pred1( X, Z) :-
  parent( X, Z).

pred1( X, Z) :-
  parent( X, Y),
  pred1( Y, Z).

% Variation a: swap clauses of the original version

pred2( X, Z) :-
  parent( X, Y),
  pred2( Y, Z).

pred2( X, Z) :-
  parent( X, Z).

% Variation b: swap goals in second clause of the original version

pred3( X, Z) :-
  parent( X, Z).

pred3( X, Z) :-
  pred3( X, Y),
  parent( Y, Z).

% Variation c: swap goals and clauses of the original version

pred4( X, Z) :-
  pred4( X, Y),
  parent( Y, Z).

pred4( X, Z) :-
  parent( X, Z).
```

Figure 2.16 Four versions of the **predecessor** program.

According to the declarative semantics of Prolog we can, without affecting the declarative meaning, change

(1) the order of clauses in the program, and
(2) the order of goals in the bodies of clauses.

The **predecessor** procedure consists of two clauses, and one of them has two goals in the body. There are, therefore, four variations of this program, all with the same declarative meaning. The four variations are obtained by

(1) swapping both clauses, and
(2) swapping the goals for each order of clauses.

The corresponding four procedures, called **pred1**, **pred2**, **pred3** and **pred4**, are shown in Figure 2.16.

There are important differences in the behaviour of these four declaratively equivalent procedures. To demonstrate these, consider the **parent** relation as shown in Figure 1.1 of Chapter 1. Now, what happens if we ask whether Tom is a predecessor of Pat using the four variations of the **predecessor** relation:

```
?- pred1( tom, pat).

yes

?- pred2( tom, pat).

yes

?- pred3( tom, pat).

yes

?- pred4( tom, pat).
```

In the last case Prolog cannot find the answer. This is manifested on the terminal by a Prolog message such as 'More core needed'.

Figure 1.11 in Chapter 1 showed the trace of **pred1** (in Chapter 1 called **predecessor**) produced for the above question. Figure 2.17 shows the corresponding traces for **pred2**, **pred3** and **pred4**. Figure 2.17(c) clearly shows that **pred4** is hopeless, and Figure 2.17(a) indicates that **pred2** is rather inefficient compared to **pred1**: **pred2** does much more searching and backtracking in the family tree.

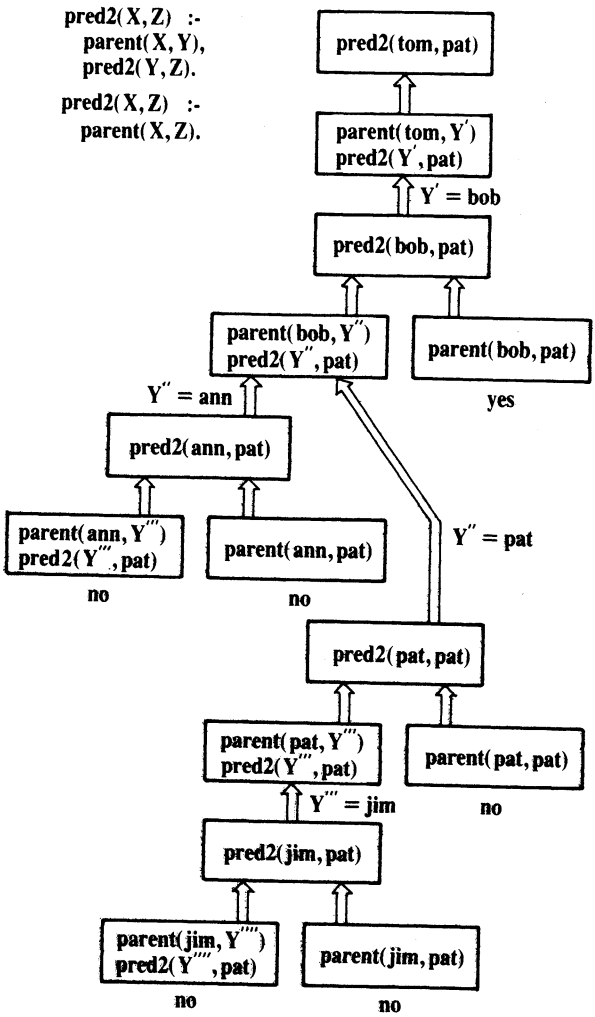
This comparison should remind us of a general practical heuristic in problem solving: it is often useful to try the simplest idea first. In our case, all the versions of the **predecessor** relation are based on two ideas:

- the simpler idea is to check whether the two arguments of the **predecessor** relation satisfy the **parent** relation;

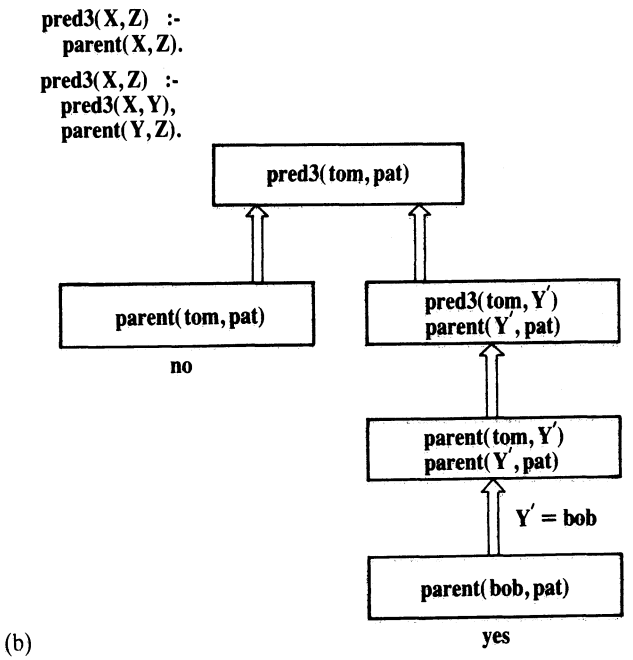
- the more complicated idea is to find somebody 'between' both people (somebody who is related to them by the **parent** and **predecessor** relations).

Of the four variations of the **predecessor** relation, **pred1** does simplest things first. On the contrary, **pred4** always tries complicated things first. **pred2** and **pred3** are in between the two extremes. Even without a detailed study of the execution traces, **pred1** should be clearly preferred merely on the grounds of the rule 'try simple things first'. This rule will be in general a useful guide in programming.

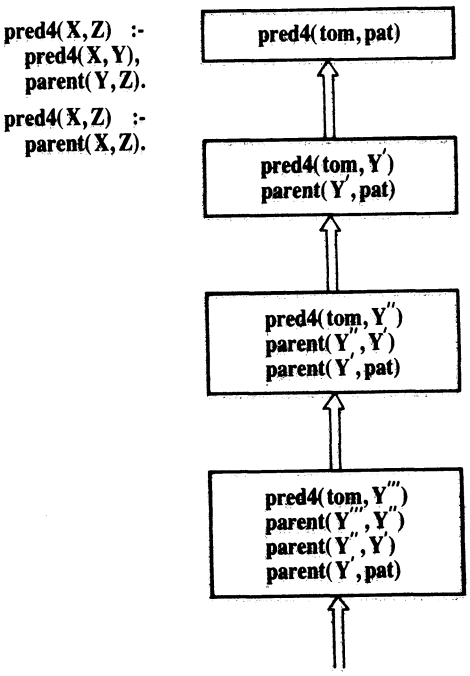
Our four variations of the **predecessor** procedure can be further compared by considering the question: What types of questions can particular variations answer, and what types can they not answer? It turns out that **pred1**



(a)



(b)



(c)

Figure 2.17 The behaviour of three formulations of the **predecessor** relation on the question: Is Tom a predecessor of Pat?

and **pred2** are both able to reach an answer for any type of question about predecessors; **pred4** can never reach an answer; and **pred3** sometimes can and sometimes cannot. One example in which **pred3** fails is:

```
?- pred3( liz, jim).
```

This question again brings the system into an infinite sequence of recursive calls. Thus **pred3** also cannot be considered procedurally correct.

2.6.3 Combining declarative and procedural views

The foregoing section has shown that the order of goals and clauses does matter. Furthermore, there are programs that are declaratively correct, but do not work in practice. Such discrepancies between the declarative and procedural meaning may appear annoying. One may argue: Why not simply forget about the declarative meaning. This argument can be brought to an extreme with a clause such as

```
predecessor( X, Z)  :-  predecessor( X, Z).
```

which is declaratively correct, but is completely useless as a working program.

The reason why we should not forget about the declarative meaning is that progress in programming technology is achieved by moving away from procedural details toward declarative aspects, which are normally easier to formulate and understand. The system itself, not the programmer, should carry the burden of filling in the procedural details. Prolog does help toward this end, although, as we have seen in this section, it only helps partially: it sometimes does work out the procedural details itself properly, and sometimes it does not. The philosophy adopted by many is that it is better to have at least *some* declarative meaning rather than *none* ('none' is the case in most other programming languages). The practical aspect of this view is that it is often rather easy to get a working program once we have a program that is declaratively correct. Consequently, a useful practical approach that often works is to concentrate on the declarative aspects of the problem, then test the resulting program on the computer, and if it fails procedurally try to rearrange the clauses and goals into a right order.

2.7 Remarks on the relation between Prolog and logic

Prolog is related to mathematical logic, so its syntax and meaning can be specified most concisely with references to logic. Prolog is indeed often defined that way. However, such an introduction to Prolog assumes that the reader is familiar with certain concepts of mathematical logic. These concepts are, on the other hand, certainly not necessary for understanding and using Prolog as a

programming tool, which is the aim of this book. For the reader who is especially interested in the relation between Prolog and logic, the following are some basic links to mathematical logic, together with some appropriate references.

Prolog's syntax is that of the *first-order predicate logic* formulas written in the so-called *clause form* (a form in which quantifiers are not explicitly written), and further restricted to *Horn clauses* only (clauses that have at most one positive literal). Clocksin and Mellish (1981) give a Prolog program that transforms a first-order predicate calculus formula into the clause form. The procedural meaning of Prolog is based on the *resolution principle* for mechanical theorem proving introduced by Robinson in his classical paper (1965). Prolog uses a special strategy for resolution theorem proving called SLD. An introduction to the first-order predicate calculus and resolution-based theorem proving can be found in Nilsson 1981. Mathematical questions regarding the properties of Prolog's procedural meaning with respect to logic are analyzed by Lloyd (1984).

Matching in Prolog corresponds to what is called *unification* in logic. However, we avoid the word unification because matching, for efficiency reasons in most Prolog systems, is implemented in a way that does not exactly correspond to unification (see Exercise 2.10). But from the practical point of view this approximation to unification is quite adequate.

Exercise

2.10 What happens if we ask Prolog:

```
?- X = f( X).
```

Should this request for matching succeed or fail? According to the definition of unification in logic this should fail, but what happens according to our definition of matching in Section 2.2? Try to explain why many Prolog implementations answer the question above with:

```
X = f(f(f(f(f(f(f(f(f(f(f(f(f( ...
```

Summary

So far we have covered a kind of basic Prolog, also called 'pure Prolog'. It is 'pure' because it corresponds closely to formal logic. Extensions whose aim is to tailor the language toward some practical needs will be covered later in the book (Chapters 3, 5, 6, 7). Important points of this chapter are:

- Simple objects in Prolog are *atoms*, *variables* and *numbers*. Structured objects, or *structures*, are used to represent objects that have several components.

- Structures are constructed by means of *functors*. Each functor is defined by its name and arity.
- The type of object is recognized entirely by its syntactic form.
- The *lexical scope* of variables is one clause. Thus the same variable name in two clauses means two different variables.
- Structures can be naturally pictured as trees. Prolog can be viewed as a language for processing trees.
- The *matching* operation takes two terms and tries to make them identical by instantiating the variables in both terms.
- Matching, if it succeeds, results in the *most general* instantiation of variables.
- The *declarative semantics* of Prolog defines whether a goal is true with respect to a given program, and if it is true, for what instantiation of variables it is true.
- A comma between goals means the conjunction of goals. A semicolon between goals means the disjunction of goals.
- The *procedural semantics* of Prolog is a procedure for satisfying a list of goals in the context of a given program. The procedure outputs the truth or falsity of the goal list and the corresponding instantiations of variables. The procedure automatically backtracks to examine alternatives.
- The declarative meaning of programs in 'pure Prolog' does not depend on the order of clauses and the order of goals in clauses.
- The procedural meaning does depend on the order of goals and clauses. Thus the order can affect the efficiency of the program; an unsuitable order may even lead to infinite recursive calls.
- Given a declaratively correct program, changing the order of clauses and goals can improve the program's efficiency while retaining its declarative correctness. Reordering is one method of preventing indefinite looping.
- There are other more general techniques, apart from reordering, to prevent indefinite looping and thereby make programs procedurally robust.
- Concepts discussed in this chapter are:

  data objects: atom, number, variable, structure
  term
  functor, arity of a functor
  principal functor of a term
  matching of terms
  most general instantiation
  declarative semantics
  instance of a clause, variant of a clause
  procedural semantics
  executing goals

References

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

Lloyd, J. W. (1984) *Foundations of Logic Programming.* Springer-Verlag.

Nilsson, N. J. (1981) *Principles of Artificial Intelligence.* Tioga; also Springer-Verlag.

Robinson, A. J. (1965) A machine-oriented logic based on the resolution principle. *JACM* **12**: 23–41.

3 Lists, Operators, Arithmetic

In this chapter we will study a special notation for lists, one of the simplest and most useful structures, and some programs for typical operations on lists. We will also look at simple arithmetic and the operator notation which often improves the readability of programs. Basic Prolog of Chapter 2, extended with these three additions, becomes a convenient framework for writing interesting programs.

3.1 Representation of lists

The *list* is a simple data structure widely used in non-numeric programming. A list is a sequence of any number of items, such as **ann**, **tennis**, **tom**, **skiing**. Such a list can be written in Prolog as:

[ann, tennis, tom, skiing]

This is, however, only the external appearance of lists. As we have already seen in Chapter 2, all structured objects in Prolog are trees. Lists are no exception to this.

How can a list be represented as a standard Prolog object? We have to consider two cases: the list is either empty or non-empty. In the first case, the list is simply written as a Prolog atom, []. In the second case, the list can be viewed as consisting of two things:

(1) the first item, called the *head* of the list;
(2) the remaining part of the list, called the *tail*.

For our example list

[ann, tennis, tom, skiing]

the head is **ann** and the tail is the list

[tennis, tom, skiing]

In general, the head can be anything (any Prolog object, for example, a tree or a variable); the tail has to be a list. The head and the tail are then combined into a structure by a special functor. The choice of this functor depends on the Prolog implementation; we will assume here that it is the dot:

.(Head, Tail)

Since **Tail** is in turn a list, it is either empty or it has its own head and tail. Therefore, to represent lists of any length no additional principle is needed. Our example list is then represented as the term:

.(ann, .(tennis, .(tom, .(skiing, []))))

Figure 3.1 shows the corresponding tree structure. Note that the empty list appears in the above term. This is because the one but last tail is a single item list:

[skiing]

This list has the empty list as its tail:

[skiing] = .(skiing, [])

This example shows how the general principle for structuring data objects in Prolog also applies to lists of any length. As our example also shows, the straightforward notation with dots and possibly deep nesting of subterms in the tail part can produce rather confusing expressions. This is the reason why Prolog provides the neater notation for lists, so that they can be written as sequences of items enclosed in square brackets. A programmer can use both notations, but the square bracket notation is, of course, normally preferred. We will be aware, however, that this is only a cosmetic improvement and that our lists will be internally represented as binary trees. When such terms are

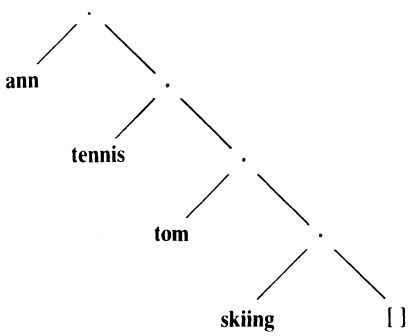


Figure 3.1 Tree representation of the list **[ann, tennis, tom, skiing]**.

output they will be automatically converted into their neater form. Thus the following conversation with Prolog is possible:

```
?- List1 = [a,b,c],
   List2 = .( a, .( b, .( c, [] ) ) ).

List1 = [a,b,c]
List2 = [a,b,c]

?- Hobbies1 = .( tennis, .( music, [] ) ),
   Hobbies2 = [ skiing, food],
   L = [ ann, Hobbies1, tom, Hobbies2].

Hobbies1 = [ tennis, music]
Hobbies2 = [ skiing, food]
L = [ ann, [tennis,music], tom, [skiing,food] ]
```

This example also reminds us that the elements of a list can be objects of any kind, in particular they can also be lists.

It is often practical to treat the whole tail as a single object. For example, let

```
L = [a,b,c]
```

Then we could write

```
Tail = [b,c]   and   L = .( a, Tail)
```

To express this in the square bracket notation for lists, Prolog provides another notational extension, the vertical bar, which separates the head and the tail:

```
L = [ a | Tail]
```

The vertical bar notation is in fact more general: we can list any number of elements followed by '|' and the list of remaining items. Thus alternative ways of writing the above list are:

```
[a,b,c] = [a | [b,c] ] = [a,b | [c] ] = [a,b,c | [] ]
```

To summarize:

- A list is a data structure that is either empty or consists of two parts: a *head* and a *tail*. The tail itself has to be a list.

- Lists are handled in Prolog as a special case of binary trees. For improved

readability Prolog provides a special notation for lists, thus accepting lists written as:

[Item1, Item2, ...]

or

[Head | Tail]

or

[Item1, Item2, ... | Others]

3.2 Some operations on lists

Lists can be used to represent sets although there is a difference: the order of elements in a set does not matter while the order of items in a list does; also, the same object can occur repeatedly in a list. Still, the most common operations on lists are similar to those on sets. Among them are:

- checking whether some object is an element of a list, which corresponds to checking for the set membership;
- concatenation of two lists, obtaining a third list, which corresponds to the union of sets;
- adding a new object to a list, or deleting some object from it.

In the remainder of this section we give programs for these and some other operations on lists.

3.2.1 Membership

Let us implement the membership relation as

member(X, L)

where X is an object and L is a list. The goal **member(X, L)** is true if X occurs in L. For example,

member(b, [a,b,c])

is true,

member(b, [a,[b,c]])

is not true, but

member([b,c], [a,[b,c]])

is true. The program for the membership relation can be based on the following observation:

X is a member of L if either
(1) X is the head of L, or
(2) X is a member of the tail of L.

This can be written in two clauses, the first is a simple fact and the second is a rule:

```
member( X, [X | Tail] ).

member( X, [Head | Tail] ) :-
  member( X, Tail).
```

3.2.2 Concatenation

For concatenating lists we will define the relation

conc(L1, L2, L3)

Here L1 and L2 are two lists, and L3 is their concatenation. For example

conc([a,b], [c,d], [a,b,c,d])

is true, but

conc([a,b], [c,d], [a,b,a,c,d])

is false. In the definition of **conc** we will have again two cases, depending on the first argument, L1:

(1) If the first argument is the empty list then the second and the third arguments must be the same list (call it L); this is expressed by the following Prolog fact:

```
conc( [], L, L).
```

(2) If the first argument of **conc** is a non-empty list then it has a head and a tail and must look like this:

[X | L1]

Figure 3.2 illustrates the concatenation of **[X | L1]** and some list L2. The result of the concatenation is the list **[X | L3]** where L3 is the concatenation of L1 and L2. In Prolog this is written as:

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

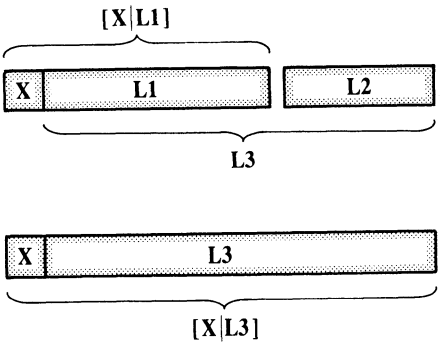


Figure 3.2 Concatenation of lists.

This program can now be used for concatenating given lists, for example:

```
?- conc( [a,b,c], [1,2,3], L).

L = [a,b,c,1,2,3]

?- conc( [a,[b,c],d], [a,[],b], L).

L = [a, [b,c], d, a, [], b]
```

Although the **conc** program looks rather simple it can be used flexibly in many other ways. For example, we can use **conc** in the inverse direction for *decomposing* a given list into two lists, as follows:

```
?- conc( L1, L2, [a,b,c] ).

L1 = []
L2 = [a,b,c];

L1 = [a]
L2 = [b,c];

L1 = [a,b]
L2 = [c];

L1 = [a,b,c]
L2 = [];

no
```

It is possible to decompose the list **[a,b,c]** in four ways, all of which were found by our program through backtracking.

We can also use our program to look for a certain pattern in a list. For

example, we can find the months that precede and the months that follow a given month, as in the following goal:

```
?- conc( Before, [may | After],
         [jan,feb,mar,apr,may,jun,jul,aug,sep,oct,nov,dec] ).

Before = [jan,feb,mar,apr]
After = [jun,jul,aug,sep,oct,nov,dec].
```

Further we can find the immediate predecessor and the immediate successor of May by asking:

```
?- conc( _, [Month1,may,Month2 | _],
         [jan,feb,mar,apr,may,jun,jul,aug,sep,oct,nov,dec] ).

Month1 = apr
Month2 = jun
```

Further still, we can, for example, delete from some list, L1, everything that follows three successive occurrences of z in L1 together with the three z's. For example:

```
?- L1 = [a,b,z,z,c,z,z,z,d,e],
   conc( L2, [z,z,z | _], L1).

L1 = [a,b,z,z,c,z,z,z,d,e]
L2 = [a,b,z,z,c]
```

We have already programmed the membership relation. Using **conc**, however, the membership relation could be elegantly programmed by the clause:

```
member1( X, L) :-
  conc( L1, [X | L2], L).
```

This clause says: X is a member of list L if L can be decomposed into two lists so that the second one has X as its head. Of course, **member1** defines the same relation as **member**. We have just used a different name to distinguish between the two implementations. Note that the above clause can be written using anonymous variables as:

```
member1( X, L) :-
  conc( _, [X | _], L).
```

It is interesting to compare both implementations of the membership relation, **member** and **member1**. **member** has a rather straightforward procedural meaning, which is as follows:

To check whether some X is a member of some list L:

(1) first check whether the head of L is equal to X, and then

(2) check whether X is a member of the tail of L.

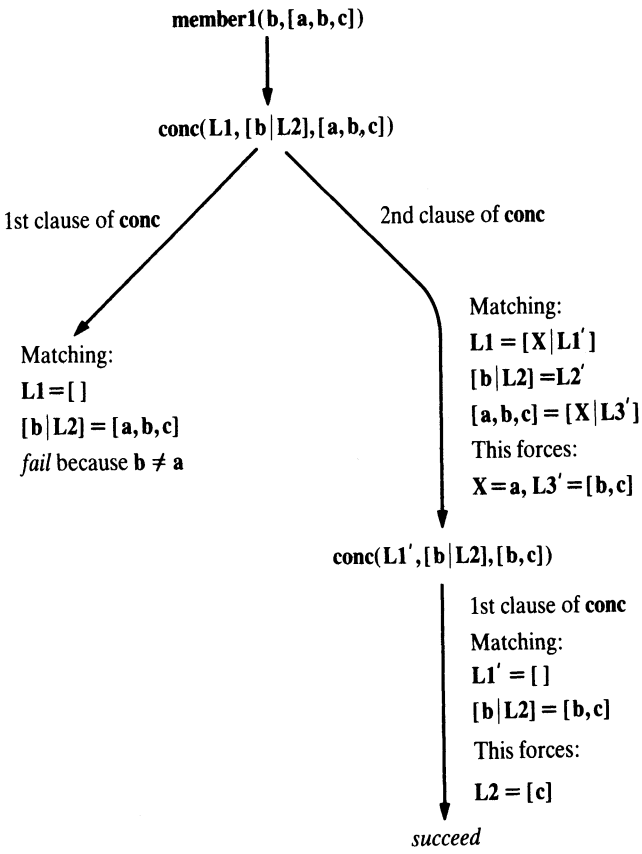


Figure 3.3 Procedure **member1** finds an item in a given list by sequentially searching the list.

On the other hand, the declarative meaning of **member1** is straightforward, but its procedural meaning is not so obvious. An interesting exercise is to find how **member1** actually computes something. An example execution trace will give some idea: let us consider the question:

```
?- member1( b, [a,b,c] ).
```

Figure 3.3 shows the execution trace. From the trace we can infer that **member1** behaves similarly to **member**. It scans the list, element by element, until the item in question is found or the list is exhausted.

Exercises

3.1 (a) Write a goal, using **conc**, to delete the last three elements from a list L producing another list L1. Hint: L is the concatenation of L1 and a three-element list.

(b) Write a sequence of goals to delete the first three elements and the last three elements from a list L producing list L2.

3.2 Define the relation

last(Item, List)

so that **Item** is the last element of a list **List**. Write two versions: (a) using the **conc** relation, (b) without **conc**.

3.2.3 Adding an item

To add an item to a list, it is easiest to put the new item in front of the list so that it becomes the new head. If X is the new item and the list to which X is added is L then the resulting list is simply

[X | L]

So we actually need no procedure for adding a new element in front of the list. Nevertheless, if we want to define such a procedure explicitly, it can be written as the fact:

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

3.2.4 Deleting an item

Deleting an item, X, from a list, L, can be programmed as a relation

del(X, L, L1)

where L1 is equal to the list L with the item X removed. The **del** relation can be defined similarly to the membership relation. We have, again, two cases:

(1) If X is the head of the list then the result after the deletion is the tail of the list.

(2) If X is in the tail then it is deleted from there.

```
del( X, [X | Tail], Tail).

del( X, [Y | Tail], [Y | Tail1] ) :-
  del( X, Tail, Tail1).
```

Like **member**, **del** is also non-deterministic in nature. If there are several occurrences of X in the list then **del** will be able to delete anyone of them by backtracking. Of course, each alternative execution will only delete one occur-

rence of X, leaving the others untouched. For example:

```
?- del( a, [a,b,a,a], L).

L = [b,a,a];

L = [a,b,a];

L = [a,b,a];

no
```

del will fail if the list does not contain the item to be deleted.

del can also be used in the inverse direction, to add an item to a list by inserting the new item anywhere in the list. For example, if we want to insert **a** at any place in the list **[1,2,3]** then we can do this by asking the question: What is L such that after deleting **a** from L we obtain **[1,2,3]**?

```
?- del( a, L, [1,2,3] ).

L = [a,1,2,3];

L = [1,a,2,3];

L = [1,2,a,3];

L = [1,2,3,a];

no
```

In general, the operation of inserting X at any place in some list **List** giving **BiggerList** can be defined by the clause:

```
insert( X, List, BiggerList) :-
  del( X, BiggerList, List).
```

In **member1** we elegantly implemented the membership relation by using **conc**. We can also use **del** to test for membership. The idea is simple: some X is a member of **List** if X can be deleted from **List**:

```
member2( X, List) :-
  del( X, List, _).
```

3.2.5 Sublist

Let us now consider the **sublist** relation. This relation has two arguments, a list L and a list S such that S occurs within L as its sublist. So

```
sublist( [c,d,e], [a,b,c,d,e,f] )
```

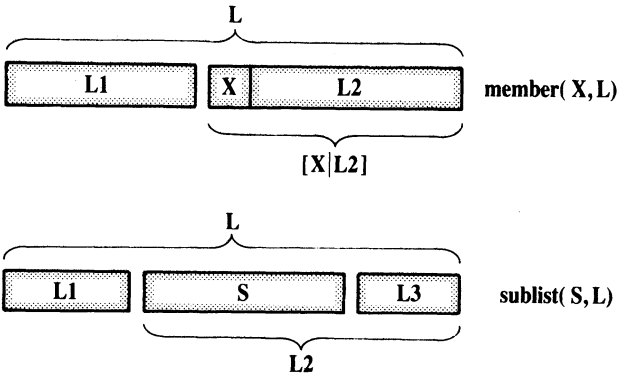


Figure 3.4 The **member** and **sublist** relations.

is true, but

```
sublist( [c,e], [a,b,c,d,e,f] )
```

is not. The Prolog program for **sublist** can be based on the same idea as **member1**, only this time the relation is more general (see Figure 3.4). Accordingly, the relation can be formulated as:

S is a sublist of L if
(1) L can be decomposed into two lists, L1 and L2, and
(2) L2 can be decomposed into two lists, S and some L3.

As we have seen before, the **conc** relation can be used for decomposing lists. So the above formulation can be expressed in Prolog as:

```
sublist( S, L) :-
  conc( L1, L2, L),
  conc( S, L3, L2).
```

Of course, the **sublist** procedure can be used flexibly in several ways. Although it was designed to check if some list occurs as a sublist within another list it can also be used, for example, to find all sublists of a given list:

```
?- sublist( S, [a,b,c] ).

S = [];

S = [a];

S = [a,b];

S = [a,b,c];

S = [b];

...
```

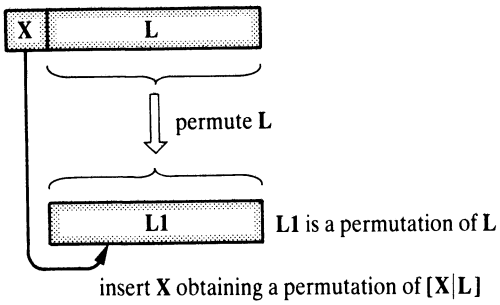


Figure 3.5 One way of constructing a permutation of the list [X | L].

3.2.6 Permutations

Sometimes it is useful to generate permutations of a given list. To this end, we will define the **permutation** relation with two arguments. The arguments are two lists such that one is a permutation of the other. The intention is to generate permutations of a list through backtracking using the **permutation** procedure, as in the following example:

```
?- permutation( [a,b,c], P).

P = [a,b,c];

P = [a,c,b];

P = [b,a,c];

...
```

The program for **permutation** can be, again, based on the consideration of two cases, depending on the first list:

(1) If the first list is empty then the second list must also be empty.

(2) If the first list is not empty then it has the form [X | L], and a permutation of such a list can be constructed as shown in Figure 3.5: first permute L obtaining L1 and then insert X at any position into L1.

Two Prolog clauses that correspond to these two cases are:

```
permutation( [], [] ).

permutation( [X | L], P) :-
  permutation( L, L1),
  insert( X, L1, P).
```

One alternative to this program would be to delete an element, X, from the first list, permute the rest of it obtaining a list P, and then add X in front of P. The corresponding program is:

```
permutation2( [], [] ).

permutation2( L, [X | P] ) :-
  del( X, L, L1),
  permutation2( L1, P).
```

It is instructive to do some experiments with our permutation programs. Its normal use would be something like this:

```
?- permutation( [red,blue,green], P).
```

This would result in all six permutations, as intended:

```
P = [ red, blue, green];

P = [ red, green, blue];

P = [ blue, red, green];

P = [ blue, green, red];

P = [ green, red, blue];

P = [ green, blue, red];

no
```

Another attempt to use **permutation** is:

```
?- permutation( L, [a,b,c] ).
```

Our first version, **permutation**, will now instantiate L successfully to all six permutations. If the user then requests more solutions, the program would never answer 'no' because it would get into an infinite loop trying to find another permutation when there is none. Our second version, **permutation2**, will in this case find only the first (identical) permutation and then immediately get into an infinite loop. Thus, some care is necessary when using these **permutation** relations.

Exercises

3.3 Define two predicates

evenlength(List) and **oddlength(List)**

so that they are true if their argument is a list of even or odd length

respectively. For example, the list **[a,b,c,d]** is 'evenlength' and **[a,b,c]** is 'oddlength'.

3.4 Define the relation

reverse(List, ReversedList)

that reverses lists. For example, **reverse([a,b,c,d], [d,c,b,a]).**

3.5 Define the predicate **palindrome(List).** A list is a palindrome if it reads the same in the forward and in the backward direction. For example, **[m,a,d,a,m].**

3.6 Define the relation

shift(List1, List2)

so that **List2** is **List1** 'shifted rotationally' by one element to the left. For example,

```
?- shift( [1,2,3,4,5], L1),
   shift( L1, L2).
```

produces:

```
L1 = [2,3,4,5,1]
L2 = [3,4,5,1,2]
```

3.7 Define the relation

translate(List1, List2)

to translate a list of numbers between 0 and 9 to a list of the corresponding words. For example:

translate([3,5,1,3], [three,five,one,three])

Use the following as an auxiliary relation:

means(0, zero). means(1, one). means(2, two). ...

3.8 Define the relation

subset(Set, Subset)

where **Set** and **Subset** are two lists representing two sets. We would like to be able to use this relation not only to check for the subset relation, but also to generate all possible subsets of a given set. For example:

```
?- subset( [a,b,c], S).

S = [a,b,c];
S = [b,c];
S = [c];
```

```
S = [];
S = [a,c];
S = [a];
...
```

3.9 Define the relation

dividelist(List, List1, List2)

so that the elements of **List** are partitioned between **List1** and **List2**, and **List1** and **List2** are of approximately the same length. For example, **partition([a,b,c,d,e], [a,c,e], [b,d]).**

3.10 Rewrite the monkey and banana program of Chapter 2 as the relation

canget(State, Actions)

to answer not just 'yes' or 'no', but to produce a sequence of monkey's actions that lead to success. Let **Actions** be such a sequence represented as a list of moves:

Actions = [walk(door,window), push(window,middle), climb, grasp]

3.11 Define the relation

flatten(List, FlatList)

where **List** can be a list of lists, and **FlatList** is **List** 'flattened' so that the elements of **List**'s sublists (or sub-sublists) are reorganized as one plain list. For example:

```
?- flatten( [a,b,[c,d],[],[[[e]]],f], L).
L = [a,b,c,d,e,f]
```

3.3 Operator notation

In mathematics we are used to writing expressions like

$$2*a + b*c$$

where + and * are operators, and 2, *a*, *b*, *c* are arguments. In particular, + and * are said to be *infix* operators because they appear *between* the two arguments. Such expressions can be represented as trees, as in Figure 3.6, and can be written as Prolog terms with + and * as functors:

```
+( *(2,a), *(b,c) )
```

Since we would normally prefer to have such expressions written in the usual,

infix style with operators, Prolog caters for this notational convenience. Prolog will therefore accept our expression written simply as:

2*a + b*c

This will be, however, only the external representation of this object, which will be automatically converted into the usual form of Prolog terms. Such a term will be output for the user, again, in its external, infix form.

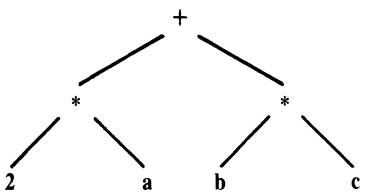


Figure 3.6 Tree representation of the expression **2*a + b*c**.

Thus expressions are dealt with in Prolog merely as a notational extension and no new principle for structuring data objects is involved. If we write **a + b**, Prolog will handle it exactly as if it had been written **+(a,b)**. In order that Prolog properly understands expressions such as **a + b*c**, Prolog has to know that * binds stronger than +. We say that + has higher precedence than *. So the precedence of operators decides what is the correct interpretation of expressions. For example, the expression **a + b*c** can be, in principle, understood either as

+(a, *(b,c))

or as

***(+(a,b), c)**

The general rule is that the operator with the highest precedence is the principal functor of the term. If expressions containing + and * are to be understood according to our normal conventions, then + has to have a higher precedence than *. Then the expression **a + b*c** means the same as **a + (b*c)**. If another interpretation is intended, then it has to be explicitly indicated by parentheses – for example, **(a + b)*c**.

A programmer can define his or her own operators. So, for example, we can define the atoms **has** and **supports** as infix operators and then write in the program facts like:

peter has information.
floor supports table.

These facts are exactly equivalent to:

```
has( peter, information).
supports( floor, table).
```

A programmer can define new operators by inserting into the program special kinds of clauses, sometimes called *directives*, which act as operator definitions. An operator definition must appear in the program before any expression containing that operator. For our example, the operator **has** can be properly defined by the directive:

```
:- op( 600, xfx, has).
```

This tells Prolog that we want to use 'has' as an operator, whose precedence is 600 and its type is 'xfx', which is a kind of infix operator. The form of the specifier 'xfx' suggests that the operator, denoted by 'f', is between the two arguments denoted by 'x'.

Notice that operator definitions do not specify any operation or action. In principle, *no operation on data is associated with an operator* (except in very special cases). Operators are normally used, as functors, only to combine objects into structures and not to invoke actions on data, although the word 'operator' appears to suggest an action.

Operator names are atoms, and their precedence must be in some range which depends on the implementation. We will assume that the range is between 1 and 1200.

There are three groups of operator types which are indicated by type specifiers such as **xfx**. The three groups are:

(1) infix operators of three types:

xfx xfy yfx

(2) prefix operators of two types:

fx fy

(3) postfix operators of two types:

xf yf

The specifiers are chosen so as to reflect the structure of the expression where 'f' represents the operator and 'x' and 'y' represent arguments. An 'f' appearing between the arguments indicates that the operator is infix. The prefix and postfix specifiers have only one argument, which follows or precedes the operator respectively.

There is a difference between 'x' and 'y'. To explain this we need to introduce the notion of the *precedence of argument*. If an argument is enclosed in parentheses or it is an unstructured object then its precedence is 0; if an argument is a structure then its precedence is equal to the precedence of its

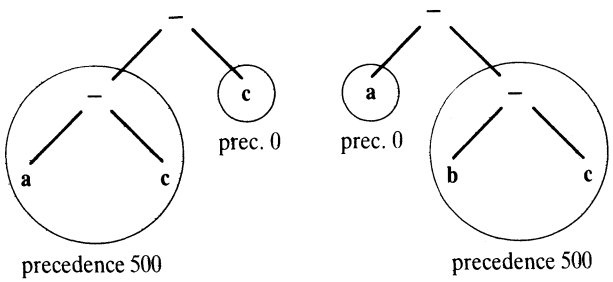


Figure 3.7 Two interpretations of the expression **a – b – c** assuming that '–' has precedence 500. If '–' is of type **yfx**, then interpretation 2 is invalid because the precedence of **b – c** is not less than the precedence of '–'.

principal functor. 'x' represents an argument whose precedence must be strictly lower than that of the operator. 'y' represents an argument whose precedence is lower or equal to that of the operator.

These rules help to disambiguate expressions with several operators of the same precedence. For example, the expression

a – b – c

is normally understood as **(a – b) – c**, and not as **a – (b – c)**. To achieve the normal interpretation the operator '–' has to be defined as **yfx**. Figure 3.7 shows why the second interpretation is then ruled out.

As another example consider the prefix operator **not**. If **not** is defined as **fy** then the expression

not not p

is legal; but if **not** is defined as **fx** then this expression is illegal because the argument to the first **not** is **not p**, which has the same precedence as **not** itself. In this case the expression has to be written with parentheses:

not (not p)

For convenience, some operators are predefined in the Prolog system so that they can be readily used and no definition is needed for them. What these operators are and what their precedences are depends on the implementation of Prolog. We will assume that this set of 'standard' operators is as if defined by the clauses in Figure 3.8. As Figure 3.8 also shows, several operators can be declared by one clause if they all have the same precedence and if they are all of the same type. In this case the operators' names are written as a list.

The use of operators can greatly improve the readability of programs. As an example let us assume that we are writing a program for manipulating

```
:- op( 1200, xfx, ':-').
:- op( 1200, fx, [ :-, ?-] ).
:- op( 1100, xfy, ';').
:- op( 1000, xfy, ',').
:- op( 700, xfx, [ =, is, < , > , =< , >= , == , =\=, \==, =:=] ).
:- op( 500, yfx, [ + , -] ).
:- op( 500, fx, [ + , -, not] ).
:- op( 400, yfx, [ *, /, div] ).
:- op( 300, xfx, mod).
```

Figure 3.8 A set of predefined operators.

Boolean expressions. In such a program we may want to state, for example, one of de Morgan's equivalence theorems, which can in mathematics be written as:

$$\sim(A \,\&\, B) \quad <===> \quad \sim A \vee \sim B$$

One way to state this in Prolog is by the clause:

```
equivalence( not( and( A, B)), or( not( A), not( B) ) ).
```

However, it is in general a good programming practice to try to retain as much resemblance as possible between the original problem notation and the notation used in the program. In our example, this can be achieved almost completely by using operators. A suitable set of operators for our purpose can be defined as:

```
:- op( 800, xfx, <===>).
:- op( 700, xfy, v).
:- op( 600, xfy, &).
:- op( 500, fy, ~).
```

Now the de Morgan's theorem can be written as the fact:

```
~(A & B) <===> ~A v ~B.
```

According to our specification of operators above, this term is understood as shown in Figure 3.9.

To summarize:

- The readability of programs can be often improved by using the operator notation. Operators can be infix, prefix or postfix.

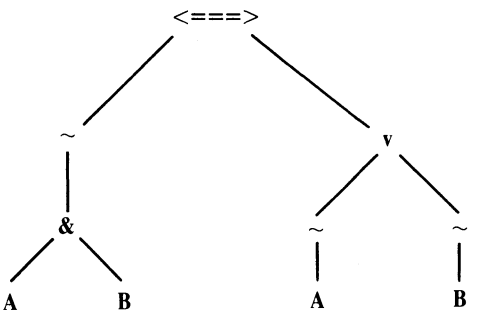


Figure 3.9 Interpretation of the term ~(A & B) <===> ~A v ~B.

- In principle, no operation on data is associated with an operator except in special cases. Operator definitions do not define any action, they only introduce new notation. Operators, as functors, only hold together components of structures.
- A programmer can define his or her own operators. Each operator is defined by its name, precedence and type.
- The precedence is an integer within some range, say between 1 and 1200. The operator with the highest precedence in the expression is the principal functor of the expression. Operators with lowest precedence bind strongest.
- The type of the operator depends on two things: (1) the position of the operator with respect to the arguments, and (2) the precedence of the arguments compared to the precedence of the operator itself. In a specifier like **xfy**, **x** indicates an argument whose precedence is strictly lower than that of the operator; **y** indicates an argument whose precedence is less than or equal to that of the operator.

Exercises

3.12 Assuming the operator definitions

```
:- op( 300, xfx, plays).
:- op( 200, xfy, and).
```

then the following two terms are syntactically legal objects:

Term1 = jimmy plays football and squash
Term2 = susan plays tennis and basketball and volleyball

How are these terms understood by Prolog? What are their principal functors and what is their structure?

3.13 Suggest an appropriate definition of operators ('was', 'of', 'the') to be able to write clauses like

```
diana was the secretary of the department.
```

and then ask Prolog:

```
?- Who was the secretary of the department.
Who = diana

?- diana was What.
What = the secretary of the department
```

3.14 Consider the program:

```
t( 0+1, 1+0).
t( X+0+1, X+1+0).
t( X+1+1, Z) :-
  t( X+1, X1),
  t( X1+1, Z).
```

How will this program answer the following questions if '+' is an infix operator of type **yfx** (as usual):

(a) `?- t( 0+1, A).`

(b) `?- t( 0+1+1, B).`

(c) `?- t( 1+0+1+1+1, C).`

(d) `?- t( D, 1+1+1+0).`

3.15 In the previous section, relations involving lists were written as:

```
member( Element, List),
conc( List1, List2, List3),
del( Element, List, NewList), ...
```

Suppose that we would prefer to write these relations as:

```
Element in List,
concatenating List1 and List2 gives List3,
deleting Element from List gives NewList, ...
```

Define 'in', 'concatenating', 'and', etc. as operators to make this possible. Also, redefine the corresponding procedures.

3.4 Arithmetic

Prolog is mainly a language for symbolic computation where the need for numerical calculation is comparatively modest. Accordingly, the means for

doing arithmetic in Prolog are also rather simple. Some of the predefined operators can be used for basic arithmetic operations. These are:

+	addition
–	subtraction
*	multiplication
/	division
mod	modulo, the remainder of integer division

Notice that this is an exceptional case in which an operator may in fact invoke an operation. But even in such cases an additional indication to perform the action will be necessary. Prolog knows how to carry out the calculation denoted by these operators, but this is not entirely sufficient for direct use. The following question is a naive attempt to request arithmetic computation:

```
?- X = 1 + 2.
```

Prolog will 'quietly' answer

```
X = 1 + 2
```

and not X = 3 as we might possibly expect. The reason is simple: the expression 1 + 2 merely denotes a Prolog term where + is the functor and 1 and 2 are its arguments. There is nothing in the above goal to force Prolog to actually activate the addition operation. A special predefined operator, **is**, is provided to circumvent this problem. The **is** operator will force evaluation. So the right way to invoke arithmetic is:

```
?- X is 1 + 2.
```

Now the answer will be:

```
X = 3
```

The addition here was carried out by a special procedure that is associated with the operator +. We call such procedures *built-in procedures*.

There is no generally agreed notational convention for arithmetic in Prolog, so different implementations of Prolog may use somewhat different notations. For example, the '/' operator may denote integer division or real division, depending on the implementation. In this book, we will assume that '/' denotes real division, and that the **div** operator denotes integer division. Accordingly, the question

```
?- X is 3/2,
   Y is 3 div 2.
```

is answered by

```
X = 1.5
Y = 1
```

The left argument of the **is** operator is a simple object. The right argument is an arithmetic expression composed of arithmetic operators, numbers and variables. Since the **is** operator will force the evaluation, all the variables in the expression must already be instantiated to numbers at the time of execution of this goal. The precedence of the predefined arithmetic operators (see Figure 3.8) is such that the associativity of arguments with operators is the same as normally in mathematics. Parentheses can be used to indicate different associations. Note that +, –, *, / and **div** are defined as **yfx**, which means that evaluation is carried out from left to right. For example,

```
X is 5 - 2 - 1
```

is interpreted as

```
X is (5 - 2) - 1
```

Arithmetic is also involved when *comparing* numerical values. We can, for example, test whether the product of 277 and 37 is greater than 10000 by the goal:

```
?- 277 * 37 > 10000.
yes
```

Note that, similarly to **is**, the '>' operator also forces the evaluation.

Suppose that we have in the program a relation **born** that relates the names of people with their birth years. Then we can retrieve the names of people born between 1950 and 1960 inclusive with the following question:

```
?- born( Name, Year),
   Year >= 1950,
   Year =< 1960.
```

The comparison operators are as follows:

X > Y	X is greater than Y
X < Y	X is less than Y
X >= Y	X is greater than or equal to Y
X =< Y	X is less than or equal to Y
X =:= Y	the values of X and Y are equal
X =\= Y	the values of X and Y are not equal

Notice the difference between the matching operators '=' and '=:='; for example, in the goals X = Y and X =:= Y. The first goal will cause the matching of the objects X and Y, and will, if X and Y match, possibly instantiate some variables in X and Y. There will be no evaluation. On the other hand, X =:= Y causes the arithmetic evaluation and cannot cause any instantiation of variables. These differences are illustrated by the following examples:

```
?- 1 + 2 =:= 2 + 1.

yes

?- 1 + 2 = 2 + 1.

no

?- 1 + A = B + 2.

A = 2
B = 1
```

Let us further illustrate the use of arithmetic operations by two simple examples. The first involves computing the greatest common divisor; the second, counting the items in a list.

Given two positive integers, X and Y, their greatest common divisor, D, can be found according to three cases:

(1) If X and Y are equal then D is equal to X.

(2) If X < Y then D is equal to the greatest common divisor of X and the difference Y – X.

(3) If Y < X then do the same as in case (2) with X and Y interchanged.

It can be easily shown by an example that these three rules actually work. Choosing, for example, X = 20 and Y = 25, the above rules would give D = 5 after a sequence of subtractions.

These rules can be formulated into a Prolog program by defining a three-argument relation, say

```
gcd( X, Y, D)
```

The three rules are then expressed as three clauses, as follows:

```
gcd( X, X, X).

gcd( X, Y, D) :-
  X < Y,
  Y1 is Y - X,
  gcd( X, Y1, D).
```

```
gcd( X, Y, D) :-
  Y < X,
  gcd( Y, X, D).
```

Of course, the last goal in the third clause could be equivalently replaced by the two goals:

```
X1 is X - Y,
gcd( X1, Y, D)
```

Our next example involves counting, which usually requires some arithmetic. An example of such a task is to establish the length of a list; that is, we have to count the items in the list. Let us define the procedure

```
length( List, N)
```

which will count the elements in a list **List** and instantiate N to their number. As was the case with our previous relations involving lists, it is useful to consider two cases:

(1) If the list is empty then its length is 0.

(2) If the list is not empty then **List = [Head | Tail]**; then its length is equal to 1 plus the length of the tail **Tail**.

These two cases correspond to the following program:

```
length( [], 0).

length( [_ | Tail], N) :-
  length( Tail, N1),
  N is 1 + N1.
```

An application of **length** can be:

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

N = 4
```

Note that in the second clause of **length**, the two goals of the body cannot be swapped. The reason for this is that N1 has to be instantiated before the goal

```
N is 1 + N1
```

can be processed. With the built-in procedure **is**, a relation has been introduced that is sensitive to the order of processing and therefore the procedural considerations have become vital.

It is interesting to see what happens if we try to program the **length** relation without the use of **is**. Such an attempt can be:

```
length1( [], 0).

length1( [_ | Tail], N) :-
  length1( Tail, N1),
  N = 1 + N1.
```

Now the goal

```
?- length1( [a,b,[c,d],e], N).
```

will produce the answer:

```
N = 1+(1+(1+(1+0))).
```

The addition was never explicitly forced and was therefore not carried out at all. But in **length1** we can, unlike in **length**, swap the goals in the second clause:

```
length1( [_ | Tail], N) :-
  N = 1 + N1,
  length1( Tail, N1).
```

This version of **length1** will produce the same result as the original version. It can also be written shorter, as follows,

```
length1( [_ | Tail], 1 + N) :-
  length1( Tail, N).
```

still producing the same result. We can, however, use **length1** to find the number of elements in a list as follows:

```
?- length1( [a,b,c], N), Length is N.

N = 1+(1+(1+0))
Length = 3
```

To summarize:

- Built-in procedures can be used for doing arithmetic.
- Arithmetic operations have to be explicitly requested by the built-in procedure **is**. There are built-in procedures associated with the predefined operators +, −, *, /, **div** and **mod**.
- At the time that evaluation is carried out, all arguments must be already instantiated to numbers.

- The values of arithmetic expressions can be compared by operators such as <, =<, etc. These operators force the evaluation of their arguments.

Exercises

3.16 Define the relation

max(X, Y, Max)

so that **Max** is the greater of two numbers X and Y.

3.17 Define the predicate

maxlist(List, Max)

so that **Max** is the greatest number in the list of numbers **List**.

3.18 Define the predicate

sumlist(List, Sum)

so that **Sum** is the sum of a given list of numbers **List**.

3.19 Define the predicate

ordered(List)

which is true if **List** is an ordered list of numbers. For example, **ordered([1,5,6,6,9,12])**.

3.20 Define the predicate

subsum(Set, Sum, SubSet)

so that **Set** is a list of numbers, **SubSet** is a subset of these numbers, and the sum of the numbers in **SubSet** is **Sum**. For example:

```
?- subsum( [1,2,5,3,2], 5, Sub).

Sub = [1,2,2];
Sub = [2,3];
Sub = [5];
...
```

3.21 Define the procedure

between(N1, N2, X)

which, for two given integers N1 and N2, generates through backtracking all the integers X that satisfy the constraint $N1 \leqq X \leqq N2$.

3.22 Define the operators 'if', 'then', 'else' and ':=' so that the following becomes a legal term:

```
if X > Y then Z := X else Z := Y
```

Choose the precedences so that 'if' will be the principal functor. Then define the relation 'if' as a small interpreter for a kind of 'if-then-else' statement of the form

```
if Val1 > Val2 then Var := Val3 else Var := Val4
```

where **Val1**, **Val2**, **Val3** and **Val4** are numbers (or variables instantiated to numbers) and **Var** is a variable. The meaning of the 'if' relation should be: if the value of **Val1** is greater than the value of **Val2** then **Var** is instantiated to **Val3**, otherwise to **Val4**. Here is an example of the use of this interpreter:

```
?- X = 2, Y = 3,
   Val2 is 2*X,
   Val4 is 4*X,
   if Y > Val2 then Z := Y else Z := Val4,
   if Z > 5 then W := 1 else W := 0.

X = 2
Y = 3
Z = 8
W = 1
```

Summary

- The list is a frequently used structure. It is either empty or consists of a *head* and a *tail* which is a list as well. Prolog provides a special notation for lists.
- Common operations on lists, programmed in this chapter, are: list membership, concatenation, adding an item, deleting an item, sublist.
- The *operator notation* allows the programmer to tailor the syntax of programs toward particular needs. Using operators the readability of programs can be greatly improved.
- New operators are defined by the directive **op**, stating the name of an operator, its type and precedence.
- In principle, there is no operation associated with an operator; operators are merely a syntactic device providing an alternative syntax for terms.
- Arithmetic is done by built-in procedures. Evaluation of an arithmetic expression is forced by the procedure **is** and by the comparison predicates <, =<, etc.

- Concepts introduced in this chapter are:

  list, head of list, tail of list
  list notation
  operators, operator notation
  infix, prefix and suffix operators
  precedence of an operator
  arithmetic built-in procedures

4 Using Structures: Example Programs

Data structures, with matching, backtracking and arithmetic, are a powerful programming tool. In this chapter we will develop the skill of using this tool through programming examples: retrieving structured information from a database, simulating a non-deterministic automaton, travel planning and eight queens on the chessboard. We will also see how the principle of data abstraction can be carried out in Prolog.

4.1 Retrieving structured information from a database

This exercise develops the skill of representing and manipulating structured data objects. It also illustrates Prolog as a natural database query language.

A database can be naturally represented in Prolog as a set of facts. For example, a database about families can be represented so that each family is described by one clause. Figure 4.1 shows how the information about each family can be structured. Each family has three components: husband, wife and children. As the number of children varies from family to family the children are represented by a list that is capable of accommodating any number of items. Each person is, in turn, represented by a structure of four components: name, surname, date of birth, job. The job information is 'unemployed',

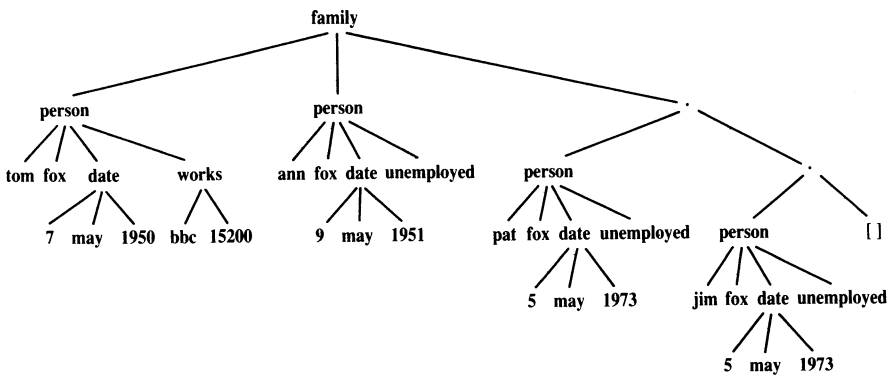


Figure 4.1 Structuring information about the family.

or it specifies the working organization and salary. The family of Figure 4.1 can be stored in the database by the clause:

```
family(
  person( tom, fox, date(7,may,1950), works(bbc,15200) ),
  person( ann, fox, date(9,may,1951), unemployed),
  [ person( pat, fox, date(5,may,1973), unemployed),
    person( jim, fox, date(5,may,1973), unemployed) ] ).
```

Our database would then be comprised of a sequence of facts like this describing all families that are of interest to our program.

Prolog is, in fact, a very suitable language for retrieving the desired information from such a database. One nice thing about Prolog is that we can refer to objects without actually specifying all the components of these objects. We can merely indicate the *structure* of objects that we are interested in, and leave the particular components in the structures unspecified or only partially specified. Figure 4.2 shows some examples. So we can refer to all Armstrong families by:

```
family( person( _, armstrong, _, _), _, _)
```

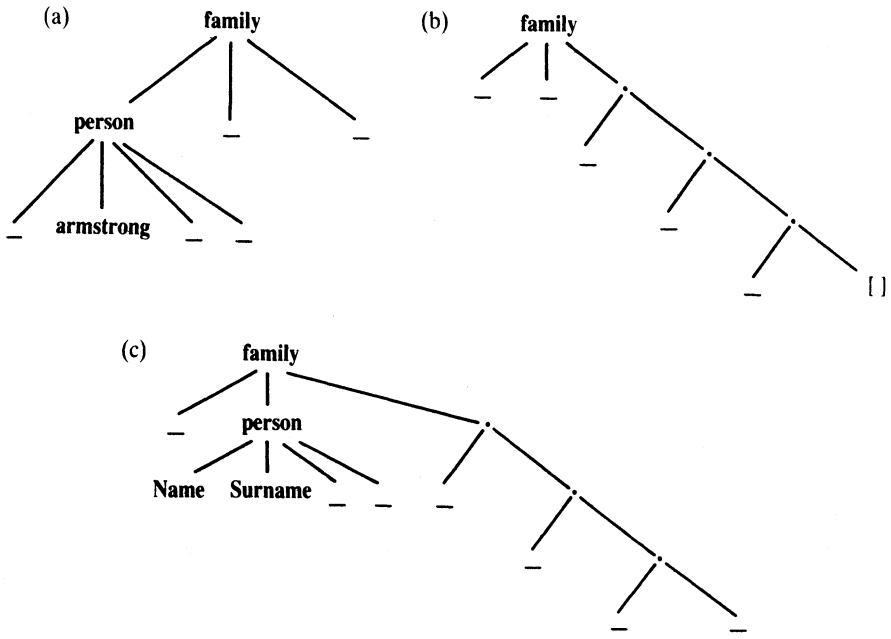


Figure 4.2 Specifying objects by their structural properties: (a) any Armstrong family; (b) any family with exactly three children; (c) any family with at least three children. Structure (c) makes provision for retrieving the wife's name through the instantiation of the variables **Name** and **Surname**.

The underscore characters denote different anonymous variables; we do not care about their values. Further, we can refer to all families with three children by the term:

```
family( _, _, [_, _, _] )
```

To find all married women that have at least three children we can pose the question:

```
?- family( _, person( Name, Surname, _, _), [_, _, _ | _] ).
```

The point of these examples is that we can specify objects of interest not by their content, but by their structure. We only indicate their structure and leave their arguments as unspecified slots.

We can provide a set of procedures that can serve as a utility to make the interaction with the database more comfortable. Such utility procedures could be part of the user interface. Some useful utility procedures for our database are:

```
husband( X) :-                          % X is a husband
  family( X, _, _).

wife( X) :-                             % X is a wife
  family( _, X, _ ).

child( X) :-                            % X is a child
  family( _, _, Children),
  member( X, Children).

member( X, [X | L] ).

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

exists( Person) :-                      % Any person in the database
  husband( Person);
  wife( Person);
  child( Person).

dateofbirth( person( _, _, Date, _), Date).

salary( person( _, _, _, works( _, S) ), S).   % Salary of working person

salary( person( _, _, _, unemployed), 0).      % Salary of unemployed
```

We can use these utilities, for example, in the following queries to the database:

- Find the names of all the people in the database:

```
?- exists( person( Name, Surname, _, _) ).
```

- Find all children born in 1981:

```
?- child( X),
   dateofbirth( X, date( _, _, 1981) ).
```

- Find all employed wives:

```
?- wife( person( Name, Surname, _, works( _, _) ) ).
```

- Find the names of unemployed people who were born before 1963:

```
?- exists( person( Name, Surname, date( _, _, Year), unemployed) ),
   Year < 1963.
```

- Find people born before 1950 whose salary is less than 8000:

```
?- exists( Person),
   dateofbirth( Person, date( _, _, Year) ),
   Year < 1950,
   salary( Person, Salary),
   Salary < 8000.
```

- Find the names of families with at least three children:

```
?- family( person( _, Name, _, _), _, [_, _, _ | _] ).
```

To calculate the total income of a family it is useful to define the sum of salaries of a list of people as a two-argument relation:

```
total( List_of_people, Sum_of_their_salaries)
```

This relation can be programmed as:

```
total( [], 0).                          % Empty list of people

total( [Person | List], Sum) :-
  salary( Person, S),                   % S: salary of first person
  total( List, Rest),                   % Rest: sum of salaries of others
  Sum is S + Rest.
```

The total income of families can then be found by the question:

```
?- family( Husband, Wife, Children),
   total( [Husband, Wife | Children], Income).
```

Let the **length** relation count the number of elements of a list, as defined in

Section 3.4. Then we can specify all families that have an income per family member of less than 2000 by:

```
?- family( Husband, Wife, Children),
   total( [Husband, Wife | Children], Income),
   length( [Husband, Wife | Children], N),        % N: size of family
   Income/N < 2000.
```

Exercises

4.1 Write queries to find the following from the family database:

(a) names of families without children;

(b) all employed children;

(c) names of families with employed wives and unemployed husbands;

(d) all the children whose parents differ in age by at least 15 years.

4.2 Define the relation

twins(Child1, Child2)

to find twins in the family database.

4.2 Doing data abstraction

Data abstraction can be viewed as a process of organizing various pieces of information into natural units (possibly hierarchically), thus structuring the information into some conceptually meaningful form. Each such unit of information should be easily accessible in the program. Ideally, all the details of implementing such a structure should be invisible to the user of the structure – the programmer can then just concentrate on objects and relations between them. The point of the process is to make the use of information possible without the programmer having to think about the details of how the information is actually represented.

Let us discuss one way of carrying out this principle in Prolog. Consider our family example of the previous section again. Each family is a collection of pieces of information. These pieces are all clustered into natural units such as a person or a family, so they can be treated as single objects. Assume again that the family information is structured as in Figure 4.1. Let us now define some relations through which the user can access particular components of a family without knowing the details of Figure 4.1. Such relations can be called *selectors* as they select particular components. The name of such a selector relation will be the name of the component to be selected. The relation will have two

arguments: first, the object that contains the component, and second, the component itself:

```
selector_relation( Object, Component_selected)
```

Here are some selectors for the family structure:

```
husband( family( Husband, _, _), Husband).

wife( family( _, Wife, _), Wife).

children( family( _, _, ChildList), ChildList).
```

We can also define selectors for particular children:

```
firstchild( Family, First) :-
  children( Family, [First | _] ).

secondchild( Family, Second) :-
  children( Family, [_, Second | _] ).

...
```

We can generalize this to selecting the Nth child:

```
nthchild( N, Family, Child) :-
  children( Family, ChildList),
  nth_member( N, ChildList, Child).          % Nth element of a list
```

Another interesting object is a person. Some related selectors according to Figure 4.1 are:

```
firstname( person( Name, _, _, _), Name).

surname( person( _, Surname, _, _), Surname).

born( person( _, _, Date, _), Date).
```

How can we benefit from selector relations? Having defined them, we can now forget about the particular way that structured information is represented. To create and manipulate this information, we just have to know the names of the selector relations and use these in the rest of the program. In the case of complicated representations, this is easier than always referring to the representation explicitly. In our family example in particular, the user does not have to know that the children are represented as a list. For example, assume that we want to say that Tom Fox and Jim Fox belong to the same family and that Jim is the second child of Tom. Using the selector relations above, we can

define two persons, call them **Person1** and **Person2**, and the family. The following list of goals does this:

```
firstname( Person1, tom), surname( Person1, fox),  % Person1 is Tom Fox
firstname( Person2, jim), surname( Person2, fox),  % Person2 is Jim Fox
husband( Family, Person1),
secondchild( Family, Person2)
```

The use of selector relations also makes programs easier to modify. Imagine that we would like to improve the efficiency of a program by changing the representation of data. All we have to do is to change the definitions of the selector relations, and the rest of the program will work unchanged with the new representation.

Exercise

4.3 Complete the definition of **nthchild** by defining the relation

nth_member(N, List, X)

which is true if X is the Nth member of **List**.

4.3 Simulating a non-deterministic automaton

This exercise shows how an abstract mathematical construct can be translated into Prolog. In addition, our resulting program will be much more flexible than initially intended.

A *non-deterministic finite automaton* is an abstract machine that reads as input a string of symbols and decides whether to *accept* or to *reject* the input string. An automaton has a number of *states* and it is always in one of the states. It can change its state by moving from the current state to another state. The internal structure of the automaton can be represented by a transition graph such as that in Figure 4.3. In this example, s_1, s_2, s_3 and s_4 are the *states* of the automaton. Starting from the initial state (s_1 in our example), the automaton moves from state to state while reading the input string. Transitions depend on the current input symbol, as indicated by the arc labels in the transition graph.

A transition occurs each time an input symbol is read. Note that transitions can be non-deterministic. In Figure 4.3, if the automaton is in state s_1 and the current input symbol is *a* then it can transit into s_1 or s_2. Some arcs are labelled *null* denoting the 'null symbol'. These arcs correspond to 'silent moves' of the automaton. Such a move is said to be *silent* because it occurs without any reading of input, and the observer, viewing the automaton as a black box, will not be able to notice that any transition has occurred.

The state s_3 is double circled, which indicates that it is a *final state*. The

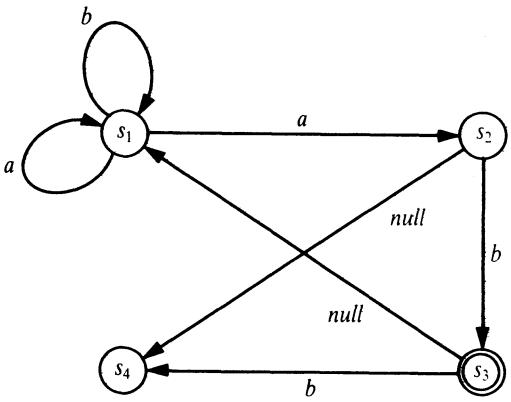


Figure 4.3 An example of a non-deterministic finite automaton.

automaton is said to *accept* the input string if there is a transition path in the graph such that

(1) it starts with the initial state,
(2) it ends with a final state, and
(3) the arc labels along the path correspond to the complete input string.

It is entirely up to the automaton to decide which of the possible moves to execute at any time. In particular, the automaton may choose to make or not to make a silent move, if it is available in the current state. But abstract non-deterministic machines of this kind have a magic property: if there is a choice then they always choose a 'right' move; that is, a move that leads to the acceptance of the input string, if such a move exists. The automaton in Figure 4.3 will, for example, accept the strings *ab* and *aabaab*, but it will reject the strings *abb* and *abba*. It is easy to see that this automaton accepts any string that terminates with *ab*, and rejects all others.

In Prolog, an automaton can be specified by three relations:

(1) A unary relation **final** which defines the final states of the automaton;

(2) A three-argument relation **trans** which defines the state transitions so that

```
trans( S1, X, S2)
```

means that a transition from a state S1 to S2 is possible when the current input symbol X is read.

(3) A binary relation

```
silent( S1, S2)
```

meaning that a silent move is possible from S1 to S2.

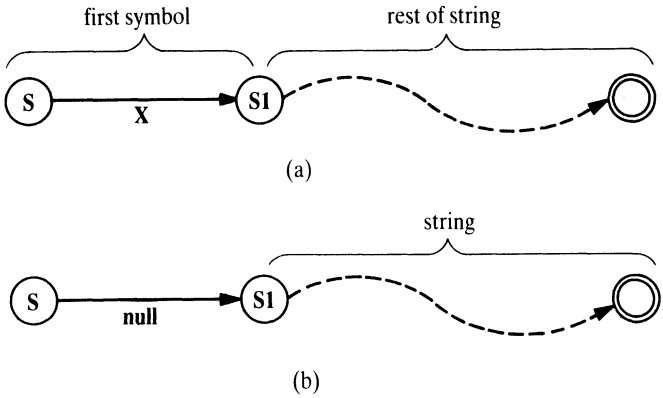


Figure 4.4 Accepting a string: (a) by reading its first symbol **X**; (b) by making a silent move.

For the automaton in Figure 4.3 these three relations are:

```
final( s3).

trans( s1, a, s1).
trans( s1, a, s2).
trans( s1, b, s1).
trans( s2, b, s3).
trans( s3, b, s4).

silent( s2, s4).
silent( s3, s1).
```

We will represent input strings as Prolog lists. So the string *aab* will be represented by **[a,a,b]**. Given the description of the automaton, the simulator will process a given input string and decide whether the string is accepted or rejected. By definition, the non-deterministic automaton accepts a given string if (starting from an initial state), after having read the whole input string, the automaton can (possibly) be in its final state. The simulator is programmed as a binary relation, **accepts**, which defines the acceptance of a string from a given state. So

```
accepts( State, String)
```

is true if the automaton, starting from the state **State** as initial state, accepts the string **String**. The **accepts** relation can be defined by three clauses. They correspond to the following three cases:

(1) The empty string, [], is accepted from a state S if S is a final state.

(2) A non-empty string is accepted from a state S if reading the first symbol in the string can bring the automaton into some state S1, and the rest of the string is accepted from S1. Figure 4.4(a) illustrates.

(3) A string is accepted from a state S if the automaton can make a silent move from S to S1 and then accept the (whole) input string from S1. Figure 4.4(b) illustrates.

These rules can be translated into Prolog as:

```
accepts( S, [] ) :-                          % Accept empty string
  final( S).

accepts( S, [X | Rest] ) :-                  % Accept by reading first symbol
  trans( S, X, S1),
  accepts( S1, Rest).

accepts( S, String) :-                       % Accept by making silent move
  silent( S, S1),
  accepts( S1, String).
```

The program can be asked, for example, about the acceptance of the string *aaab* by:

```
?- accepts( s1, [a,a,a,b] ).

yes
```

As we have already seen, Prolog programs are often able to solve more general problems than problems for which they were originally developed. In our case, we can also ask the simulator which state our automaton can be in initially so that it will accept the string *ab*:

```
?- accepts( S, [a,b] ).

S = s1;

S = s3
```

Amusingly, we can also ask: What are all the strings of length 3 that are accepted from state s_1?

```
?- accepts( s1, [X1,X2,X3] ).

X1 = a
X2 = a
X3 = b;

X1 = b
X2 = a
X3 = b;

no
```

If we prefer the acceptable input strings to be typed out as lists then we can formulate the question as:

```
?- String = [_, _, _], accepts( s1, String).

String = [a,a,b];

String = [b,a,b];

no
```

We can make further experiments asking even more general questions, such as: From what states will the automaton accept input strings of length 7?

Further experimentation could involve modifications in the structure of the automaton by changing the relations **final**, **trans** and **silent**. The automaton in Figure 4.3 does not contain any cyclic 'silent path' (a path that consists only of silent moves). If in Figure 4.3 a new transition

```
silent( s1, s3)
```

is added then a 'silent cycle' is created. But our simulator may now get into trouble. For example, the question

```
?- accepts( s1, [a] ).
```

would induce the simulator to cycle in state s_1 indefinitely, all the time hoping to find some way to the final state.

Exercises

4.4 Why could cycling not occur in the simulation of the original automaton in Figure 4.3, when there was no 'silent cycle' in the transition graph?

4.5 Cycling in the execution of **accepts** can be prevented, for example, by counting the number of moves made so far. The simulator would then be requested to search only for paths of some limited length. Modify the **accepts** relation this way. Hint: Add a third argument: the maximum number of moves allowed:

```
accepts( State, String, Max_moves)
```

4.4 Travel planning

In this section we will construct a program that gives advice on planning air travel. The program will be a rather simple advisor, yet it will be able to answer

some useful questions, such as:

- What days of the week is there a direct flight from London to Ljubljana?
- How can I get from Ljubljana to Edinburgh on Thursday?
- I have to visit Milan, Ljubljana and Zurich, starting from London on Tuesday and returning to London on Friday. In what sequence should I visit these cities so that I have no more than one flight each day of the tour?

The program will be centred around a database holding the flight information. This will be represented as a three-argument relation

timetable(Place1, Place2, List_of_flights)

where **List_of_flights** is a list of structured items of the form:

Departure_time / Arrival_time / Flight_number / List_of_days

List_of_days is either a list of weekdays or the atom 'alldays'. One clause of the **timetable** relation can be, for example:

```
timetable( london, edinburgh,
           [  9:40 / 10:50 / ba4733 / alldays,
             19:40 / 20:50 / ba4833 / [mo,tu,we,th,fr,su] ] ).
```

The times are represented as structured objects with two components, hours and minutes, combined by the operator ':'.

The main problem is to find exact routes between two given cities on a given day of the week. This will be programmed as a four-argument relation:

route(Place1, Place2, Day, Route)

Here **Route** is a sequence of flights that satisfies the following criteria:

(1) the start point of the route is **Place1**;
(2) the end point is **Place2**;
(3) all the flights are on the same day of the week, **Day**;
(4) all the flights in **Route** are in the **timetable** relation;
(5) there is enough time for transfer between flights.

The route is represented as a list of structured objects of the form:

From - To : Flight_number : Departure_time

We will also use the following auxiliary predicates:

(1) **flight(Place1, Place2, Day, Flight_num, Dep_time, Arr_time)**

This says that there is a flight, **Flight_num**, between **Place1** and **Place2** on the day of the week **Day** with the specified departure and arrival times.

(2) **deptime(Route, Time)**

Departure time of **Route** is **Time**.

(3) **transfer(Time1, Time2)**

There is at least 40 minutes between **Time1** and **Time2**, which should be sufficient for transfer between two flights.

The problem of finding a route is reminiscent of the simulation of the non-deterministic automaton of the previous section. The similarities of both problems are as follows:

- The states of the automaton correspond to the cities.
- A transition between two states corresponds to a flight between two cities.
- The **transition** relation of the automaton corresponds to the **timetable** relation.
- The automaton simulator finds a path in the transition graph between the initial state and a final state; the travel planner finds a route between the start city and the end city of the tour.

Not surprisingly, therefore, the **route** relation can be defined similarly to the **accepts** relation, with the exception that here we have no 'silent moves'. We have two cases:

(1) Direct flight connection: if there is a direct flight between places **Place1** and **Place2** then the route consists of this flight only:

```
route( Place1, Place2, Day, [Place1-Place2 : Fnum : Dep] ) :-
  flight( Place1, Place2, Day, Fnum, Dep, Arr).
```

(2) Indirect flight connection: the route between places P1 and P2 consists of the first flight, from P1 to some intermediate place P3, followed by a route between P3 to P2. In addition, there must be enough time between the arrival of the first flight and the departure of the second flight for transfer.

```
route( P1, P2, Day, [P1-P3 : Fnum1 : Dep1 | Route] ) :-
  route( P3, P2, Day, Route),
  flight( P1, P3, Day, Fnum1, Dep1, Arr1),
  deptime( Route, Dep2),
  transfer( Arr1, Dep2).
```

The auxiliary relations **flight**, **transfer** and **deptime** are easily programmed and are included in the complete travel planning program in Figure 4.5. Also included is an example timetable database.

Our route planner is extremely simple and may examine paths that obviously lead nowhere. Yet it will suffice if the flight database is not large. A really large database would require more intelligent planning to cope with the large number of potential candidate paths.

Some example questions to the program are as follows:

- What days of the week is there a direct flight from London to Ljubljana?

```
?- flight( london, ljubljana, Day, _, _, _).

Day = fr;

Day = su;

no
```

```
% A FLIGHT ROUTE PLANNER

:- op( 50, xfy, :).

flight( Place1, Place2, Day, Fnum, Deptime, Arrtime) :-
  timetable( Place1, Place2, Flightlist),
  member( Deptime / Arrtime / Fnum / Daylist , Flightlist),
  flyday( Day, Daylist).

member( X, [X | L] ).

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

flyday( Day, Daylist) :-
  member( Day, Daylist).

flyday( Day, alldays) :-
  member( Day, [mo,tu,we,th,fr,sa,su] ).

route( P1, P2, Day [P1-P2 : Fnum : Deptime] ) :-          % Direct flight
  flight( P1, P2, Day, Fnum, Deptime, _).

route( P1, P2, Day, [P1-P3 : Fnum1 : Dep1 | Route] ) :- % Indirect connection
  route( P3, P2, Day, Route),
  flight( P1, P3, Day, Fnum1, Dep1, Arr1),
  deptime( Route, Dep2),
  transfer( Arr1, Dep2).

deptime( [P1-P2 : Fnum : Dep | _], Dep).

transfer( Hours1:Mins1, Hours2:Mins2) :-
  60 * (Hours2 - Hours1) + Mins2 - Mins1 >= 40.
```

```
% A FLIGHT DATABASE

timetable( edinburgh, london,
           [  9:40 / 10:50 / ba4733 / alldays,
             13:40 / 14:50 / ba4773 / alldays,
             19:40 / 20:50 / ba4833 / [mo,tu,we,th,fr,su] ] ).

timetable( london, edinburgh,
           [  9:40 / 10:50 / ba4732 / alldays,
             11:40 / 12:50 / ba4752 / alldays,
             18:40 / 19:50 / ba4822 / [mo,tu,we,th,fr] ] ).

timetable( london, ljubljana,
           [ 13:20 / 16:20 / ju201 / [fr],
             13:20 / 16:20 / ju213 / [su] ] ).

timetable( london, zurich,
           [  9:10 / 11:45 / ba614 / alldays,
             14:45 / 17:20 / sr805 / alldays ] ).

timetable( london, milan,
           [  8:30 / 11:20 / ba510 / alldays,
             11:00 / 13:50 / az459 / alldays ] ).

timetable( ljubljana, zurich,
           [ 11:30 / 12:40 / ju322 / [tu,th] ] ).

timetable( ljubljana, london,
           [ 11:10 / 12:20 / yu200 / [fr],
             11:25 / 12:20 / yu212 / [su] ] ).

timetable( milan, london,
           [  9:10 / 10:00 / az458 / alldays,
             12:20 / 13:10 / ba511 / alldays ] ).

timetable( milan, zurich,
           [  9:25 / 10:15 / sr621 / alldays,
             12:45 / 13:35 / sr623 / alldays ] ).

timetable( zurich, ljubljana,
           [ 13:30 / 14:40 / yu323 / [tu,th] ] ).

timetable( zurich, london,
           [  9:00 / 9:40 / ba613 / [mo,tu,we,th,fr,sa],
             16:10 / 16:55 / sr806 / [mo,tu,we,th,fr,su] ] ).

timetable( zurich, milan,
           [  7:55 / 8:45 / sr620 / alldays ] ).
```

Figure 4.5 A flight route planner and an example flight timetable.

- How can I get from Ljubljana to Edinburgh on Thursday?

```
?- route( ljubljana, edinburgh, th, R).

R = [ljubljana-zurich:yu322:11:30, zurich-london:sr806:16:10,
     london-edinburgh:ba4822:18:40]
```

- How can I visit Milan, Ljubljana and Zurich, starting from London on Tuesday and returning to London on Friday, with no more than one flight each day of the tour? This question is somewhat trickier. It can be formulated by using the **permutation** relation, programmed in Chapter 3. We are asking for a permutation of the cities Milan, Ljubljana and Zurich such that the corresponding flights are possible on successive days:

```
?- permutation( [milan, ljubljana, zurich], [City1, City2, City3] ),
   flight( london, City1, tu, FN1, Dep1, Arr1),
   flight( City1, City2, we, FN2, Dep2, Arr2),
   flight( City2, City3, th, FN3, Dep3, Arr3),
   flight( City3, london, fr, FN4, Dep4, Arr4).

City1 = milan
City2 = zurich
City3 = ljubljana
FN1 = ba510
Dep1 = 8:30
Arr1 = 11:20
FN2 = sr621
Dep2 = 9:25
Arr2 = 10:15
FN3 = yu323
Dep3 = 13:30
Arr3 = 14:40
FN4 = yu200
Dep4 = 11:10
Arr4 = 12:20
```

4.5 The eight queens problem

The problem here is to place eight queens on the empty chessboard in such a way that no queen attacks any other queen. The solution will be programmed as a unary predicate

```
solution( Pos)
```

which is true if and only if **Pos** represents a position with eight queens that do

not attack each other. It will be interesting to compare various ideas for programming this problem. Therefore we will present three programs based on somewhat different representations of the problem.

4.5.1 Program 1

First we have to choose a representation of the board position. One natural choice is to represent the position by a list of eight items, each of them corresponding to one queen. Each item in the list will specify a square of the board on which the corresponding queen is sitting. Further, each square can be specified by a pair of coordinates (X and Y) on the board, where each coordinate is an integer between 1 and 8. In the program we can write such a pair as

X/Y

where, of course, the '/' operator is not meant to indicate division, but simply combines both coordinates together into a square. Figure 4.6 shows one solution of the eight queens problem and its list representation.

Having chosen this representation, the problem is to find such a list of the form

[X1/Y1, X2/Y2, X3/Y3, ..., X8/Y8]

which satisfies the no-attack requirement. Our procedure **solution** will have to search for a proper instantiation of the variables **X1, Y1, X2, Y2, ..., X8, Y8**. As we know that all the queens will have to be in different columns to prevent vertical attacks, we can immediately constrain the choice and so make the search task easier. We can thus fix the X-coordinates so that the solution list will fit the following, more specific template:

[1/Y1, 2/Y2, 3/Y3, ..., 8/Y8]

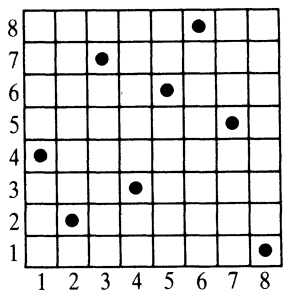


Figure 4.6 A solution to the eight queens problem. This position can be specified by the list **[1/4, 2/2, 3/7, 4/3, 5/6, 6/8, 7/5, 8/8]**.

We are interested in the solution on a board of size 8 by 8. However, in programming, in general, the key to the solution is often in considering a more general problem. Paradoxically, it is often the case that the solution for the more general problem is easier to formulate than that for the more specific, original problem; then the original problem is simply solved as a special case of the more general problem.

The creative part of the problem is to find the correct generalization of the original problem. In our case, a good idea is to generalize the number of queens (the number of columns in the list) from 8 to any number, including zero. The **solution** relation can then be formulated by considering two cases:

Case 1 The list of queens is empty: the empty list is certainly a solution because there is no attack.

Case 2 The list of queens is non-empty: then it looks like this:

[X/Y | Others]

In case 2, the first queen is at some square **X/Y** and the other queens are at squares specified by the list **Others**. If this is to be a solution then the following conditions must hold:

(1) There must be no attack between the queens in the list **Others**; that is, **Others** itself must also be a solution.

(2) X and Y must be integers between 1 and 8.

(3) A queen at square **X/Y** must not attack any of the queens in the list **Others**.

To program the first condition we can simply use the **solution** relation itself. The second condition can be specified as follows: Y will have to be a member of the list of integers between 1 and 8 – that is, **[1,2,3,4,5,6,7,8]**. On the other hand, we do not have to worry about X since the solution list will have to match the template in which the X-coordinates are already specified. So X will be guaranteed to have a proper value between 1 and 8. We can implement the third condition as another relation, **noattack**. All this can then be written in Prolog as follows:

```
solution( [X/Y | Others] ) :-
  solution( Others),
  member( Y, [1,2,3,4,5,6,7,8] ),
  noattack( X/Y, Others).
```

It now remains to define the **noattack** relation:

```
noattack( Q, Qlist)
```

Again, this can be broken down into two cases:

(1) If the list **Qlist** is empty then the relation is certainly true because there is no queen to be attacked.

(2) If **Qlist** is not empty then it has the form **[Q1 | Qlist1]** and two conditions must be satisfied:

   (a) the queen at Q must not attack the queen at Q1, and

   (b) the queen at Q must not attack any of the queens in **Qlist1**.

To specify that a queen at some square does not attack another square is easy: the two squares must not be in the same row, the same column or the same diagonal. Our solution template guarantees that all the queens are in different columns, so it only remains to specify explicitly that:

- the Y-coordinates of the queens are different, and
- they are not in the same diagonal, either upward or downward; that is, the distance between the squares in the X-direction must not be equal to that in the Y-direction.

Figure 4.7 shows the complete program. To alleviate its use a template list has

```
solution( [] ).

solution( [X/Y | Others] ) :-          % First queen at X/Y, other queens at Others
  solution( Others),
  member( Y, [1,2,3,4,5,6,7,8] ),
  noattack( X/Y, Others).              % First queen does not attack others

noattack( _, [] ).                     % Nothing to attack

noattack( X/Y, [X1/Y1 | Others] ) :-
  Y =\= Y1,                            % Different Y-coordinates
  Y1-Y =\= X1-X,                       % Different diagonals
  Y1-Y =\= X-X1,
  noattack( X/Y, Others).

member( X, [X | L] ).

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

% A solution template

template( [1/Y1,2/Y2,3/Y3,4/Y4,5/Y5,6/Y6,7/Y7,8/Y8] ).
```

Figure 4.7 Program 1 for the eight queens problem.

been added. This list can be retrieved in a question for generating solutions. So we can now ask

```
?- template( S), solution( S).
```

and the program will generate solutions as follows:

```
S = [ 1/4, 2/2, 3/7, 4/3, 5/6, 6/8, 7/5, 8/1];
S = [ 1/5, 2/2, 3/4, 4/7, 5/3, 6/8, 7/6, 8/1];
S = [ 1/3, 2/5, 3/2, 4/8, 5/6, 6/4, 7/7, 8/1];
...
```

Exercise

4.6 When searching for a solution, the program of Figure 4.7 explores alternative values for the Y-coordinates of the queens. At which place in the program is the order of alternatives defined? How can we easily modify the program to change the order? Experiment with different orders with the view of studying the executional efficiency of the program.

4.5.2 Program 2

In the board representation of program 1, each solution had the form

```
[ 1/Y1, 2/Y2, 3/Y3, ..., 8/Y8]
```

because the queens were simply placed in consecutive columns. No information is lost if the X-coordinates were omitted. So a more economical representation of the board position can be used, retaining only the Y-coordinates of the queens:

```
[ Y1, Y2, Y3, ..., Y8]
```

To prevent the horizontal attacks, no two queens can be in the same row. This imposes a constraint on the Y-coordinates. The queens have to occupy all the rows 1, 2, ..., 8. The choice that remains is the *order* of these eight numbers. Each solution is therefore represented by a permutation of the list

```
[1,2,3,4,5,6,7,8]
```

Such a permutation, S, is a solution if all the queens are safe. So we can write:

```
solution( S) :-
  permutation( [1,2,3,4,5,6,7,8], S),
  safe( S).
```

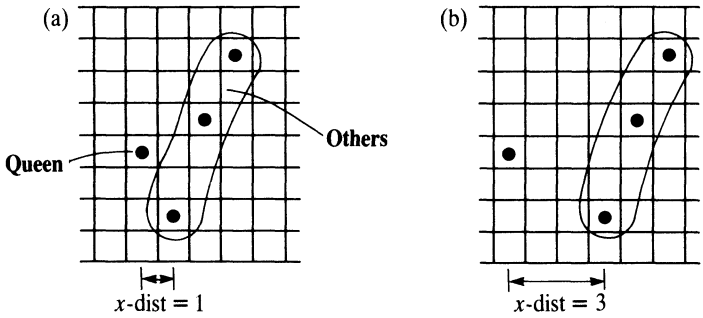


Figure 4.8 (a) X-distance between **Queen** and **Others** is 1. (b) X-distance between **Queen** and **Others** is 3.

We have already programmed the **permutation** relation in Chapter 3, but the **safe** relation remains to be specified. We can split its definition into two cases:

(1) S is the empty list: this is certainly safe as there is nothing to be attacked.

(2) S is a non-empty list of the form **[Queen | Others]**. This is safe if the list **Others** is safe, and **Queen** does not attack any queen in the list **Others**.

In Prolog, this is:

```
safe( [] ).

safe( [Queen | Others] ) :-
  safe( Others),
  noattack( Queen, Others).
```

The **noattack** relation here is slightly trickier. The difficulty is that the queens' positions are only defined by their Y-coordinates, and the X-coordinates are not explicitly present. This problem can be circumvented by a small generalization of the **noattack** relation, as illustrated in Figure 4.8. The goal

```
noattack( Queen, Others)
```

is meant to ensure that **Queen** does not attack **Others** when the X-distance between **Queen** and **Others** is equal to 1. What is needed is the generalization of the X-distance between **Queen** and **Others**. So we add this distance as the third argument of the **noattack** relation:

```
noattack( Queen, Others, Xdist)
```

Accordingly, the **noattack** goal in the **safe** relation has to be modified to

```
noattack( Queen, Others, 1)
```

```
solution( Queens) :-
  permutation( [1,2,3,4,5,6,7,8], Queens),
  safe( Queens).

permutation( [], [] ).

permutation( [Head | Tail], PermList) :-
  permutation( Tail, PermTail),
  del( Head, PermList, PermTail).          % Insert Head in permuted Tail

del( A, [A | List], List).

del( A, [B | List], [B | List1] ) :-
  del( A, List, List1).

safe( [] ).

safe( [Queen | Others] ) :-
  safe( Others),
  noattack( Queen, Others, 1).

noattack( _, [], _).

noattack( Y, [Y1 | Ylist], Xdist) :-
  Y1-Y =\= Xdist,
  Y-Y1 =\= Xdist,
  Dist1 is Xdist + 1,
  noattack( Y, Ylist, Dist1).
```

Figure 4.9 Program 2 for the eight queens problem.

The **noattack** relation can now be formulated according to two cases, depending on the list **Others**: if **Others** is empty then there is no target and certainly no attack; if **Others** is non-empty then **Queen** must not attack the first queen in **Others** (which is **Xdist** columns from **Queen**) and also the tail of **Others** at **Xdist + 1**. This leads to the program shown in Figure 4.9.

4.5.3 Program 3

Our third program for the eight queens problem will be based on the following reasoning. Each queen has to be placed on some square; that is, into some column, some row, some upward diagonal and some downward diagonal. To make sure that all the queens are safe, each queen must be placed in a different column, a different row, a different upward and a different downward diago-

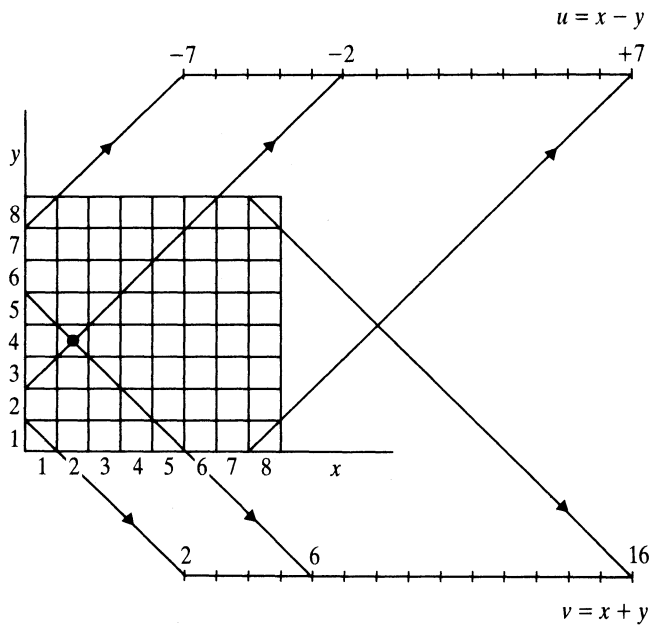


Figure 4.10 The relation between columns, rows, upward and downward diagonals. The indicated square has coordinates: $x = 2$, $y = 4$, $u = 2 - 4 = -2$, $v = 2 + 4 = 6$.

nal. It is thus natural to consider a richer representation system with four coordinates:

x	columns
y	rows
u	upward diagonals
v	downward diagonals

The coordinates are not independent: given x and y, u and v are determined (Figure 4.10 illustrates). For example, as

$$u = x - y$$

$$v = x + y$$

The domains for all four dimensions are:

Dx = [1,2,3,4,5,6,7,8]

Dy = [1,2,3,4,5,6,7,8]

Du = [-7,-6,-5,-4,-3,-2,-1,0,1,2,3,4,5,6,7]

Dv = [2,3,4,5,6,7,8,9,10,11,12,13,14,15,16]

The eight queens problem can now be stated as follows: select eight 4-tuples (X,Y,U,V) from the domains (X from Dx, Y from Dy, etc.), never using the same element twice from any of the domains. Of course, once X and Y are chosen, U and V are determined. The solution can then be, roughly speaking, as follows: given all four domains, select the position of the first queen, delete the corresponding items from the four domains, and then use the rest of the domains for placing the rest of the queens. A program based on this idea is shown in Figure 4.11. The board position is, again, represented by a list of Y-coordinates. The key relation in this program is

sol(Ylist, Dx, Dy, Du, Dv)

which instantiates the Y-coordinates (in **Ylist**) of the queens, assuming that they are placed in consecutive columns taken from Dx. All Y-coordinates and the corresponding U and V-coordinates are taken from the lists Dy, Du and Dv. The top procedure, **solution**, can be invoked by the question

?- solution(S).

This will cause the invocation of **sol** with the complete domains that correspond to the problem space of eight queens.

```
solution( Ylist) :-
  sol( Ylist,                                  % Y-coordinates of queens
       [1,2,3,4,5,6,7,8],                      % Domain for Y-coordinates
       [1,2,3,4,5,6,7,8],                      % Domain for X-coordinates
       [-7,-6,-5,-4,-3,-2,-1,0,1,2,3,4,5,6,7], % Upward diagonals
       [2,3,4,5,6,7,8,9,10,11,12,13,14,15,16] ).  % Downward diagonals

sol( [], [], Dy, Du, Dv).

sol( [Y | Ylist], [X | Dx1], Dy, Du, Dv) :-
  del( Y, Dy, Dy1),                            % Choose a Y-coordinate
  U is X-Y,                                    % Corresponding upward diagonal
  del( U, Du, Du1),                            % Remove it
  V is X+Y,                                    % Corresponding downward diagonal
  del( V, Dv, Dv1),                            % Remove it
  sol( Ylist, Dx1, Dy1, Du1, Dv1).             % Use remaining values

del( A, [A | List], List).

del( A, [B | List], [B | List1] ) :-
  del( A, List, List1).
```

Figure 4.11 Program 3 for the eight queens problem.

The **sol** procedure is general in the sense that it can be used for solving the N-queens problem (on a chessboard of size N by N). It is only necessary to properly set up the domains Dx, Dy, etc.

It is practical to mechanize the generation of the domains. For that we need a procedure

```
gen( N1, N2, List)
```

which will, for two given integers N1 and N2, produce the list

```
List = [ N1, N1 + 1, N1 + 2, ..., N2 - 1, N2]
```

Such a procedure is:

```
gen( N, N, [N] ).

gen( N1, N2, [N1 | List] ) :-
  N1 < N2,
  M is N1 + 1,
  gen( M, N2, List).
```

The top level relation, **solution**, has to be accordingly generalized to

```
solution( N, S)
```

where N is the size of the board and S is a solution represented as a list of Y-coordinates of N queens. The generalized **solution** relation is:

```
solution( N, S) :-
  gen( 1, N, Dxy),
  Nu1 is 1 - N, Nu2 is N - 1,
  gen( Nu1, Nu2, Du),
  Nv2 is N + N,
  gen( 2, Nv2, Dv),
  sol( S, Dxy, Dxy, Du, Dv).
```

For example, a solution to the 12-queens problem would be generated by:

```
?- solution( 12, S).

S = [1,3,5,8,10,12,6,11,2,7,9,4]
```

4.5.4 Concluding remarks

The three solutions to the eight queens problem show how the same problem can be approached in different ways. We also varied the representation of data. Sometimes the representation was more economical, sometimes it was more

explicit and partially redundant. The drawback of the more economical representation is that some information always has to be recomputed when it is required.

At several points, the key step toward the solution was to generalize the problem. Paradoxically, by considering a more general problem, the solution became easier to formulate. This generalization principle is a kind of standard technique that can often be applied.

Of the three programs, the third one illustrates best how to approach general problems of constructing under constraints a structure from a given set of elements.

A natural question is: Which of the three programs is most efficient? In this respect, program 2 is far inferior while the other two programs are similar. The reason is that permutation-based program 2 constructs complete permutations while the other two programs are able to recognize and reject unsafe permutations when they are only partially constructed. Program 3 is the most efficient. It avoids some of the arithmetic computation that is essentially captured in the redundant board representation this program uses.

Exercise

4.7 Let the squares of the chessboard be represented by pairs of their coordinates of the form **X/Y**, where both X and Y are between 1 and 8.

(a) Define the relation **jump(Square1, Square2)** according to the knight jump on the chessboard. Assume that **Square1** is always instantiated to a square while **Square2** can be uninstantiated. For example:

```
?- jump( 1/1, S).

S = 3/2;

S = 2/3;

no
```

(b) Define the relation **knightpath(Path)** where **Path** is a list of squares that represent a legal path of a knight on the empty chessboard.

(c) Using this **knightpath** relation, write a question to find any knight's path of length 4 moves from square 2/1 to the opposite edge of the board (Y = 8) that goes through square 5/4 after the second move.

Summary

The examples of this chapter illustrate some strong points and characteristic features of Prolog programming:

- A database can be naturally represented as a set of Prolog facts.

- Prolog's mechanisms of querying and matching can be flexibly used for retrieving structured information from a database. In addition, utility procedures can be easily defined to further alleviate the interaction with a particular database.
- *Data abstraction* can be viewed as a programming technique that makes the use of complex data structures easier, and contributes to the clarity of programs. It is easy in Prolog to carry out the essential principles of data abstraction.
- Abstract mathematical constructs, such as automata, can often be readily translated into executable Prolog definitions.
- As in the case of eight queens, the same problem can be approached in different ways by varying the representation of the problem. Often, introducing redundancy into the representation saves computation. This entails trading space for time.
- Often, the key step toward a solution is to generalize the problem. Paradoxically, by considering a more general problem the solution may become easier to formulate.

5 Controlling Backtracking

We have already seen that a programmer can control the execution of a program through the ordering of clauses and goals. In this chapter we will look at another control facility, called 'cut', for preventing backtracking.

5.1 Preventing backtracking

Prolog will automatically backtrack if this is necessary for satisfying a goal. Automatic backtracking is a useful programming concept because it relieves the programmer of the burden of programming backtracking explicitly. On the other hand, uncontrolled backtracking may cause inefficiency in a program. Therefore we sometimes want to control, or to prevent, backtracking. We can do this in Prolog by using the 'cut' facility.

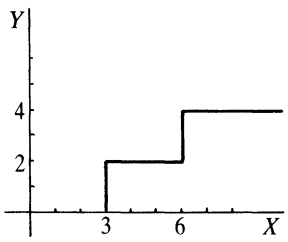


Figure 5.1 A double-step function.

Let us first study the behaviour of a simple example program whose execution involves some unnecessary backtracking. We will identify those points at which the backtracking is useless and leads to inefficiency.

Consider the double-step function shown in Figure 5.1. The relation between X and Y can be specified by three rules:

Rule 1: if $X < 3$ then $Y = 0$

Rule 2: if $3 \leqq X$ and $X < 6$ then $Y = 2$

Rule 3: if $6 \leqq X$ then $Y = 4$

This can be written in Prolog as a binary relation

```
f( X, Y)
```

as follows:

```
f( X, 0) :- X < 3.                % Rule 1
f( X, 2) :- 3 =< X, X < 6.        % Rule 2
f( X, 4) :- 6 =< X.               % Rule 3
```

This program, of course, assumes that before **f(X, Y)** is executed X is already instantiated to a number, as this is required by the comparison operators.

We will make two experiments with this program. Each experiment will reveal some source of inefficiency in the program, and we will remove each source in turn by using the cut mechanism.

5.1.1 Experiment 1

Let us analyze what happens when the following question is posed:

```
?- f( 1, Y),   2 < Y.
```

When executing the first goal, **f(1, Y)**, Y becomes instantiated to 0. So the second goal becomes

```
2 < 0
```

which fails, and so does the whole goal list. This is straightforward, but before admitting that the goal list is not satisfiable, Prolog tries, through backtracking, two useless alternatives. The detailed trace is shown in Figure 5.2.

The three rules about the **f** relation are mutually exclusive so that one of them at most will succeed. Therefore we, not Prolog, know that as soon as one rule succeeds there is no point in trying to use the others, as they are bound to fail. In the example of Figure 5.2, rule 1 has become known to succeed at the point indicated by 'CUT'. In order to prevent futile backtracking at this point we have to tell Prolog explicitly *not* to backtrack. We can do this by using the cut mechanism. The 'cut' is written as ! and is inserted between goals as a kind of pseudo-goal. Our program, rewritten with cuts, is:

```
f( X, 0) :- X < 3, !.
f( X, 2) :- 3 =< X, X < 6, !.
f( X, 4) :- 6 =< X.
```

The ! symbol will now prevent backtracking at the points that it appears in the

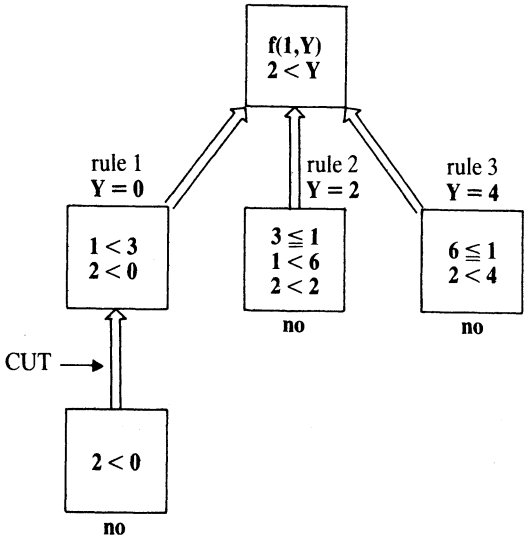


Figure 5.2 At the point marked 'CUT' we already know that the rules 2 and 3 are bound to fail.

program. If we now ask

```
?- f( 1, Y), 2 < Y.
```

Prolog will produce the same left-hand branch as in Figure 5.2. This branch will fail at the goal $2 < 0$. Now Prolog will try to backtrack, but not beyond the point marked ! in the program. The alternative branches that correspond to 'rule 2' and 'rule 3' will not be generated.

The new program, equipped with cuts, is in general more efficient than the original version without cuts. When the execution fails, the new program will in general recognize this sooner than the original program.

To conclude, we have improved the efficiency by adding cuts. If the cuts are now removed in this example, the program will still produce the same result; it will perhaps only spend more time. It can be said that, in our case, by introducing the cut we only changed the procedural meaning of the program; that is, the declarative meaning was not affected. We will see later that using a cut may affect the declarative meaning as well.

5.1.2 Experiment 2

Let us now perform a second experiment with the second version of our program. Suppose we ask:

```
?- f( 7, Y).
Y = 4
```

Let us analyze what has happened. All three rules were tried before the answer was obtained. This produced the following sequence of goals:

Try rule 1: $7 < 3$ fails, backtrack and try rule 2 (cut was not reached)

Try rule 2: $3 \leqq 7$ succeeds, but then $7 < 6$ fails, backtrack and try rule 3 (cut was not reached)

Try rule 3: $6 \leqq 7$ succeeds

This trace reveals another source of inefficiency. First it is established that $X < 3$ is not true ($7 < 3$ fails). The next goal is $3 =< X$ ($3 \leqq 7$ succeeds). But we know that once the first test has failed the second test is bound to succeed as it is the negation of the first. Therefore the second test is redundant and the corresponding goal can be omitted. The same is true about the goal $6 =< X$ in rule 3. This leads to the following, more economical formulation of the three rules:

if $X < 3$ then $Y = 0$,
otherwise if $X < 6$ then $Y = 2$,
otherwise $Y = 4$.

We can now omit the conditions in the program that are guaranteed to be true whenever they are executed. This leads to the third version of the program:

```
f( X, 0) :- X < 3, !.

f( X, 2) :- X < 6, !.

f( X, 4).
```

This program produces the same results as our original version, but is more efficient than both previous versions. But what happens if we *now* remove the cuts? The program becomes:

```
f( X, 0) :- X < 3.

f( X, 2) :- X < 6.

f( X, 4).
```

This may produce multiple solutions some of which are not correct. For example:

```
?- f( 1, Y).

Y = 0;

Y = 2;

Y = 4;

no
```

It is important to notice that, in contrast to the second version of the program, this time the cuts do not only affect the procedural behaviour, but also change the declarative meaning of the program.

A more precise meaning of the cut mechanism is as follows:

> Let us call the 'parent goal' the goal that matched the head of the clause containing the cut. When the cut is encountered as a goal it succeeds immediately, but it commits the system to all choices made between the time the 'parent goal' was invoked and the time the cut was encountered. All the remaining alternatives between the parent goal and the cut are discarded.

To clarify this definition consider a clause of the form:

H :- B1, B2, ..., Bm, !, ..., Bn.

Let us assume that this clause was invoked by a goal G that matched H. Then G is the parent goal. At the moment that the cut is encountered, the system has already found some solution of the goals **B1, ..., Bm**. When the cut is executed, this (current) solution of **B1, ..., Bm** becomes frozen and all possible remaining alternatives are discarded. Also, the goal G now becomes committed to this clause: any attempt to match G with the head of some other clause is precluded.

Let us apply these rules to the following example:

```
C :- P, Q, R, !, S, T, U.

C :- V.

A :- B, C, D.

?- A.
```

Here A, B, C, D, P, etc. have the syntax of terms. The cut will affect the execution of the goal C in the following way. Backtracking will be possible within the goal list P, Q, R; however, as soon as the cut is reached, all alternative solutions of the goal list P, Q, R are suppressed. The alternative clause about C,

```
C :- V.
```

will also be discarded. However, backtracking will still be possible within the goal list S, T, U. The 'parent goal' of the clause containing the cut is the goal C in the clause

```
A :- B, C, D.
```

Therefore the cut will only affect the execution of the goal C. On the other

hand, it will be 'invisible' from goal A. So automatic backtracking within the goal list B, C, D will remain active regardless of the cut within the clause used for satisfying C.

5.2 Examples using cut

5.2.1 Computing maximum

The procedure for finding the larger of two numbers can be programmed as a relation

```
max( X, Y, Max)
```

where Max = X if X is greater than or equal to Y, and Max is Y if X is less than Y. This corresponds to the following two clauses:

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

These two rules are mutually exclusive. If the first one succeeds then the second one will fail. If the first one fails then the second must succeed. Therefore a more economical formulation, with 'otherwise', is possible:

If $X \geqq Y$ then Max = X,
otherwise Max = Y.

This is written in Prolog using a cut as:

```
max( X, Y, X) :- X >= Y, !.
max( X, Y, Y).
```

5.2.2 Single-solution membership

We have been using the relation

```
member( X, L)
```

for establishing whether X is in list L. The program was:

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

This is 'non-deterministic': if X occurs several times then any occurrence can be found. Let us now change **member** into a deterministic procedure which will find only the first occurrence. The change is simple: we only have to prevent backtracking as soon as X is found, which happens when the first clause succeeds. The modified program is:

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

This program will generate just one solution. For example:

```
?- member( X, [a,b,c] ).
X = a;
no
```

5.2.3 Adding an element to a list without duplication

Often we want to add an item X to a list L so that X is added only if X is not yet in L. If X is already in L then L remains the same because we do not want to have redundant duplicates in L. The **add** relation has three arguments

```
add( X, L, L1)
```

where X is the item to be added, L is the list to which X is to be added and L1 is the resulting new list. Our rule for adding can be formulated as:

If X is a member of list L then L1 = L,
otherwise L1 is equal to L with X inserted.

It is easiest to insert X in front of L so that X becomes the head of L1. This is then programmed as follows:

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

The behaviour of this procedure is illustrated by the following example:

```
?- add( a, [b,c], L).
L = [a,b,c]
?- add( X, [b,c], L).
L = [b,c]
X = b
```

```
?- add( a, [b,c,X], L).

L = [b,c,a]
X = a
```

This example is instructive because we cannot easily program the 'non-duplicate adding' without the use of cut or another construct derived from the cut. If we omit the cut in the foregoing program then the **add** relation will also add duplicate items. For example:

```
?- add( a, [a,b,c], L).

L = [a,b,c];

L = [a,a,b,c]
```

So the cut is necessary here to specify the right relation, and not only to improve efficiency. The next example also illustrates this point.

5.2.4 Classification into categories

Assume we have a database of results of tennis games played by members of a club. The pairings were not arranged in any systematic way, so each player just played some other players. The results are in the program represented as facts like:

```
beat( tom, jim).
beat( ann, tom).
beat( pat, jim).
```

We want to define a relation

```
class( Player, Category)
```

that ranks the players into categories. We have just three categories:

winner: every player who won all his or her games is a winner

fighter: any player that won some games and lost some

sportsman: any player who lost all his or her games

For example, if all the results available are just those above then Ann and Pat are winners, Tom is a fighter and Jim is a sportsman.

It is easy to specify the rule for a fighter:

```
X is a fighter if
  there is some Y such that X beat Y and
  there is some Z such that Z beat X.
```

Now a rule for a winner:

X is a winner if
  X beat some Y and
  X was not beaten by anybody.

This formulation contains 'not' which cannot be directly expressed with our present Prolog facilities. So the formulation of **winner** appears trickier. The same problem occurs with **sportsman**. The problem can be circumvented by combining the definition of **winner** with that of **fighter**, and using the 'otherwise' connective. Such a formulation is:

If X beat somebody and X was beaten by somebody
  then X is a fighter,
  otherwise if X beat somebody
    then X is a winner,
    otherwise if X got beaten by somebody
      then X is a sportsman.

This formulation can be readily translated into Prolog. The mutual exclusion of the three alternative categories is indicated by the cuts:

```
class( X, fighter) :-
  beat( X, _),
  beat( _, X), !.

class( X, winner) :-
  beat( X, _), !.

class( X, sportsman) :-
  beat( _, X).
```

Exercises

5.1 Let a program be:

```
p( 1).
p( 2) :- !.
p( 3).
```

Write all Prolog's answers to the following questions:

(a) ?- p(X).

(b) ?- p(X), p(Y).

(c) ?- p(X), !, p(Y).

5.2 The following relation classifies numbers into three classes: positive, zero and negative:

```
class( Number, positive) :- Number > 0.
class( 0, zero).
class( Number, negative) :- Number < 0.
```

Define this procedure in a more efficient way using cuts.

5.3 Define the procedure

```
split( Numbers, Positives, Negatives)
```

which splits a list of numbers into two lists: positive ones (including zero) and negative ones. For example,

```
split( [3,-1,0,5,-2], [3,0,5], [-1,-2] )
```

Propose two versions: one with a cut and one without.

5.3 Negation as failure

'Mary likes all animals but snakes'. How can we say this in Prolog? It is easy to express one part of this statement: Mary likes any X if X is an animal. This is in Prolog:

```
likes( mary, X) :- animal( X).
```

But we have to exclude snakes. This can be done by using a different formulation:

> If X is a snake then 'Mary likes X' is not true,
> otherwise if X is an animal then Mary likes X.

That something is not true can be said in Prolog by using a special goal, **fail**, which always fails, thus forcing the parent goal to fail. The above formulation is translated into Prolog, using **fail**, as follows:

```
likes( mary, X) :-
  snake( X), !, fail.

likes( mary, X) :-
  animal( X).
```

The first rule here will take care of snakes: if X is a snake then the cut will prevent backtracking (thus excluding the second rule) and **fail** will cause the

failure. These two clauses can be written more compactly as one clause:

```
likes( mary, X) :-
  snake( X), !, fail;
  animal( X).
```

We can use the same idea to define the relation

different(X, Y)

which is true if X and Y are different. We have to be more precise, however, because 'different' can be understood in several ways:

- X and Y are not literally the same;
- X and Y do not match;
- the values of arithmetic expressions X and Y are not equal.

Let us choose here that X and Y are different if they do not match. The key to saying this in Prolog is:

If X and Y match then **different(X, Y)** fails,
otherwise **different(X, Y)** succeeds.

We again use the cut and **fail** combination:

```
different( X, X) :- !, fail.
different( X, Y).
```

This can also be written as one clause:

```
different( X, Y) :-
  X = Y, !, fail;
  true.
```

true is a goal that always succeeds.

These examples indicate that it would be useful to have a unary predicate 'not' such that

not(Goal)

is true if **Goal** is not true. We will now define the **not** relation as follows:

If **Goal** succeeds then **not(Goal)** fails,
otherwise **not(Goal)** succeeds.

This definition can be written in Prolog as:

```
not( P) :-
  P, !, fail;
  true.
```

Henceforth, we will assume that **not** is a built-in Prolog procedure that behaves as defined here. We will also assume that **not** is defined as a prefix operator, so that we can also write the goal

```
not( snake(X) )
```

as:

```
not snake( X)
```

Many Prolog implementations do in fact support this notation. If not, then we can always define **not** ourselves.

It should be noted that **not** defined as failure, as here, does not exactly correspond to negation in mathematical logic. This difference can cause unexpected behaviour if **not** is used without care. This will be discussed later in the chapter.

Nevertheless, **not** is a useful facility and can often be used advantageously in place of cut. Our two examples can be rewritten with **not** as:

```
likes( mary, X) :-
  animal( X),
  not snake( X).

different( X, Y) :-
  not ( X = Y).
```

This certainly looks better than our original formulations. It is more natural and is easier to read.

Our tennis classification program of the previous section can also be rewritten, using **not**, in a way that is closer to the initial definition of the three categories:

```
class( X, fighter) :-
  beat( X, _),
  beat( _, X).

class( X, winner) :-
  beat( X, _),
  not beat( _, X).
```

```
class( X, sportsman) :-
  beat( _, X),
  not beat( X, _).
```

As another example of the use of **not** let us reconsider program 1 for the eight queens problem of the previous chapter (Figure 4.7). We specified the **no_attack** relation between a queen and other queens. This relation can be formulated also as the negation of the attack relation. Figure 5.3 shows a program modified accordingly.

Exercises

5.4 Given two lists, **Candidates** and **RuledOut**, write a sequence of goals (using **member** and **not**) that will through backtracking find all the items in **Candidates** that are not in **RuledOut**.

5.5 Define the set subtraction relation

difference(Set1, Set2, SetDifference)

where all the three sets are represented as lists. For example:

difference([a,b,c,d], [b,d,e,f], [a,c])

```
solution( [] ).

solution( [X/Y | Others] ) :-
  solution( Others),
  member( Y, [1,2,3,4,5,6,7,8] ),
  not attacks( X/Y, Others).

attacks( X/Y, Others) :-
  member( X1/Y1, Others),
  ( Y1 = Y;
    Y1 is Y + X1 - X;
    Y1 is Y - X1 + X ).

member( A, [A | L] ).

member( A, [B | L] ) :-
  member( A, L).

% Solution template

template( [1/Y1,2/Y2,3/Y3,4/Y4,5/Y5,6/Y6,7/Y7,8/Y8] ).
```

Figure 5.3 Another eight queens program.

5.6 Define the predicate

unifiable(List1, Term, List2)

where **List2** is the list of all the members of **List1** that match **Term**, but are not instantiated by this matching. For example:

?- unifiable([X, b, t(Y)], t(a), List]).

List = [X, t(Y)]

Note that X and Y have to remain uninstantiated although the matching with t(a) does cause their instantiation. Hint: Use **not (Term1 = Term2)**. If **Term1 = Term2** succeeds then **not (Term1 = Term2)** fails and the resulting instantiation is undone!

5.4 Problems with cut and negation

Using the cut facility we get something, but not for nothing. The advantages and disadvantages of using cut were illustrated by examples in the previous sections. Let us summarize, first the advantages:

(1) With cut we can often improve the efficiency of the program. The idea is to explicitly tell Prolog: do not try other alternatives because they are bound to fail.

(2) Using cut we can specify mutually exclusive rules; so we can express rules of the form:

if condition P *then* conclusion Q,
otherwise conclusion R

In this way, cut enhances the expressive power of the language.

The reservations against the use of cut stem from the fact that we can lose the valuable correspondence between the declarative and procedural meaning of programs. If there is no cut in the program we can change the order of clauses and goals, and this will only affect the efficiency of the program, not the declarative meaning. On the other hand, in programs with cuts, a change in the order of clauses may affect the declarative meaning. This means that we can get different results. The following example illustrates:

```
p :- a, b.
p :- c.
```

The declarative meaning of this program is: p is true if and only if a and b are both true or c is true. This can be written as a logic formula:

p <===> (a & b) v c

We can change the order of the two clauses and the declarative meaning remains the same. Let us now insert a cut:

```
p :- a, !, b.
p :- c.
```

The declarative meaning is now:

p <===> (a & b) v (~a & c)

If we swap the clauses,

```
p :- c.
p :- a, !, b.
```

then the meaning becomes:

p <===> c v (a & b)

The important point is that when we use the cut facility we have to pay more attention to the procedural aspects. Unfortunately, this additional difficulty increases the probability of a programming error.

In our examples in the previous sections we have seen that sometimes the removal of a cut from the program can change the declarative meaning of the program. But there were also cases in which the cut had no effect on the declarative meaning. The use of cuts of the latter type is less delicate, and therefore cuts of this kind are sometimes called 'green cuts'. From the point of view of readability of programs, green cuts are 'innocent' and their use is quite acceptable. When reading a program, green cuts can simply be ignored.

On the contrary, cuts that do affect the declarative meaning are called 'red cuts'. Red cuts are the ones that make programs hard to understand, and they should be used with special care.

Cut is often used in combination with a special goal, **fail**. In particular, we defined the negation of a goal (**not**) as the failure of the goal. The negation, so defined, is just a special (more restricted) way of using cut. For reasons of clarity we will prefer to use **not** instead of the *cut–fail* combination (whenever possible), because the negation is a higher level concept and is intuitively clearer than the *cut–fail* combination.

It should be noted that **not** may also cause problems, and so should also be used with care. The problem is that **not**, as defined here, does not correspond exactly to negation in mathematics. If we ask Prolog

```
?- not human( mary).
```

Prolog will probably answer 'yes'. But this should not be understood as Prolog

saying 'Mary is not human'. What Prolog really means to say is: 'There is not enough information in the program to prove that Mary is human'. This arises because when processing a **not** goal, Prolog does not try to prove this goal directly. Instead, it tries to prove the opposite, and if the opposite cannot be proved then Prolog assumes that the **not** goal succeeds. Such reasoning is based on the so-called *closed world assumption*. According to this assumption *the world is closed* in the sense that everything that exists is in the program or can be derived from the program. Accordingly then, if something is not in the program (or cannot be derived from it) then it is not true and consequently its negation is true. This deserves special care because we do not normally assume that 'the world is closed': with not explicitly entering the clause

```
human( mary).
```

into our program, we do not normally mean to imply that Mary is not human.

We will, by example, further study the special care that **not** requires:

```
r( a).

q( b).

p( X)  :-  not r( X).
```

If we now ask

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

then Prolog will answer

```
X = b
```

If we ask apparently the same question

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

then Prolog will answer:

```
no
```

The reader is invited to trace the program to understand why we get different answers. The key difference between both questions is that the variable X is, in the first case, already instantiated when **p(X)** is executed, whereas at that point X is not yet instantiated in the second case.

We have discussed problems with cut, which also indirectly occur in **not**, in detail. The intention has been to warn users about the necessary care, not to definitely discourage the use of cut. Cut is useful and often necessary. And after all, the kind of complications that are incurred by cut in Prolog commonly occur when programming in other languages as well.

Summary

- The cut facility prevents backtracking. It is used both to improve the efficiency of programs and to enhance the expressive power of the language.
- Efficiency is improved by explicitly telling Prolog (with cut) not to explore alternatives that we know are bound to fail.
- Cut makes it possible to formulate mutually exclusive conclusions through rules of the form:

  if **Condition** *then* **Conclusion1** *otherwise* **Conclusion2**

- Cut makes it possible to introduce *negation as failure*: **not Goal** is defined through the failure of **Goal**.
- Two special goals are sometimes useful: **true** always succeeds, **fail** always fails.
- There are also some reservations against cut: inserting a cut may destroy the correspondence between the declarative and procedural meaning of a program. Therefore, it is part of good programming style to use cut with care and not to use it without reason.
- **not** defined through failure does not exactly correspond to negation in mathematical logic. Therefore, the use of **not** also requires special care.

Reference

The distinction between 'green cuts' and 'red cuts' was proposed by van Emden (1982).

van Emden, M. (1982) Red and green cuts. *Logic Programming Newsletter*: **2**.

6 Input and Output

In this chapter we will investigate some built-in facilities for reading data from computer files and for outputting data to files. These procedures can also be used for formatting data objects in the program to achieve a desired external representation of these objects. We will also look at facilities for reading programs and for constructing and decomposing atoms.

6.1 Communication with files

The method of communication between the user and the program that we have been using up to now consists of user questions to the program and program answers in terms of instantiations of variables. This method of communication is simple and practical and, in spite of its simplicity, suffices to get the information in and out. However, it is often not quite sufficient because it is too rigid. Extensions to this basic communication method are needed in the following areas:

- input of data in forms other than questions – for example, in the form of English sentences
- output of information in any format desired
- input from and output to any computer file and not just the user terminal

Built-in predicates aimed at these extensions depend on the implementation of Prolog. We will study here a simple and handy repertoire of such predicates, which is part of many Prolog implementations. However, the implementation manual should be consulted for details and specificities.

We will first consider the question of directing input and output to files, and then how data can be input and output in different forms.

Figure 6.1 shows a general situation in which a Prolog program communicates with several files. The program can, in principle, read data from several input files, also called *input streams*, and output data to several output files, also called *output streams*. Data coming from the user's terminal is treated as just another input stream. Data output to the terminal is, analogously, treated as another output stream. Both of these 'pseudo-files' are referred to by the

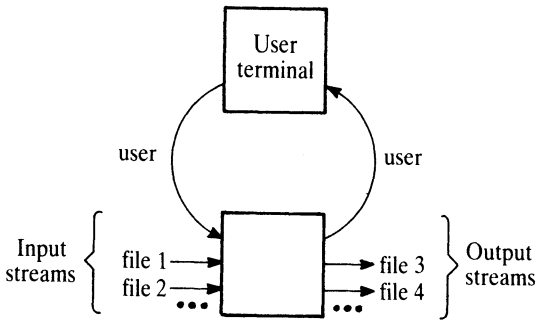


Figure 6.1 Communication between a Prolog program and several files.

name **user**. The names of other files can be chosen by the programmer according to the rules for naming files in the computer system used.

At any time during the execution of a Prolog program, only two files are 'active': one for input and one for output. These two files are called the *current input stream* and the *current output stream* respectively. At the beginning of execution these two streams correspond to the user's terminal. The current input stream can be changed to another file, **Filename**, by the goal

see(Filename)

Such a goal succeeds (unless there is something wrong with **Filename**) and causes, as a side effect, that input to be switched from the previous input stream to **Filename**. So a typical example of using the **see** predicate is the following sequence of goals, which reads something from **file1** and then switches back to the terminal:

```
...
see( file1),
read_from_file( Information),
see( user),
...
```

The current output stream can be changed by a goal of the form:

tell(Filename)

A sequence of goals to output some information to **file3**, and then redirect succeeding output back to the terminal, is:

```
...
tell( file3),
write_on_file( Information),
tell( user),
...
```

The goal

seen

closes the current input file. The goal

told

closes the current output file.

Files can only be processed sequentially. In this sense all files behave in the same way as the terminal. Each request to read something from an input file will cause reading at the current position in the current input stream. After the reading, the current position will be, of course, moved to the next unread item. So the next request for reading will start reading at this new current position. If a request for reading is made at the end of a file, then the information returned by such a request is the atom **end_of_file**. Once some information has been read, it is not possible to reread it again.

Writing is similar; each request to output information will append this information at the end of the current output stream. It is not possible to move backward and to overwrite part of the file.

All files are 'text-files' – that is, files of characters. Characters are letters, digits and special characters. Some of them are said to be non-printable because when they are output on the terminal they do not appear on the screen. They may, however, have other effects, such as spacing between columns and lines.

There are two main ways in which files can be viewed in Prolog, depending on the form of information. One way is to consider the character as the basic element of the file. Accordingly, one input or output request will cause a single character to be read or written. The built-in predicates for this are **get**, **get0** and **put**.

The other way of viewing a file is to consider bigger units of information as basic building blocks of the file. Such a natural bigger unit is the Prolog term. So each input/output request of this type would transfer a whole term from the current input stream or to the current output stream respectively. Predicates for transfer of terms are **read** and **write**. Of course, in this case, the information in the file has to be in a form that is consistent with the syntax of terms.

What kind of file organization is chosen will, of course, depend on the problem. Whenever the problem specification will allow the information to be naturally squeezed into the syntax of terms, we will prefer to use a file of terms. It will then be possible to transfer a whole meaningful piece of information with a single request. On the other hand, there are problems whose nature dictates some other organization of files. An example is the processing of natural language sentences, say, to generate a dialogue in English between the system and the user. In such cases, files will have to be viewed as sequences of characters that cannot be parsed into terms.

6.2 Processing files of terms

6.2.1 *read* and *write*

The built-in predicate **read** is used for reading terms from the current input stream. The goal

```
read( X)
```

will cause the next term, T, to be read, and this term will be matched with X. If X is a variable then, as a result, X will become instantiated to T. If matching does not succeed then the goal **read(X)** fails. The predicate **read** is deterministic, so in the case of failure there will be no backtracking to input another term. Each term in the input file must be followed by a full stop and a space or carriage-return.

If **read(X)** is executed when the end of the current input file has been reached then X will become instantiated to the atom **end_of_file**.

The built-in predicate **write** outputs a term. So the goal

```
write( X)
```

will output the term X on the current output file. X will be output in the same standard syntactic form in which Prolog normally displays values of variables. A useful feature of Prolog is that the **write** procedure 'knows' to display any term no matter how complicated it may be.

There are additional built-in predicates for formatting the output. They insert spaces and new lines into the output stream. The goal

```
tab( N)
```

causes N spaces to be output. The predicate **nl** (which has no arguments) causes the start of a new line at output.

The following examples will illustrate the use of these procedures.

Let us assume that we have a procedure that computes the cube of a number:

```
cube( N, C) :-
  C is N * N * N.
```

Suppose we want to use this for calculating the cubes of a sequence of numbers. We could do this by a sequence of questions:

```
?- cube( 2, X).
X = 8
```

```
?- cube( 5, Y).

Y = 125

?- cube( 12, Z).

Z = 1728
```

For each number, we had to type in the corresponding goal. Let us now modify this program so that the **cube** procedure will read the data itself. Now the program will keep reading data and outputting their cubes until the atom **stop** is read:

```
cube :-
  read( X),
  process( X).

process( stop) :- !.

process( N) :-
  C is N * N * N,
  write( C),
  cube.
```

This is an example of a program whose declarative meaning is awkward to formulate. However, its procedural meaning is straightforward: to execute **cube**, first read X and then process it; if **X = stop** then everything has been done, otherwise write the cube of X and recursively call the **cube** procedure to process further data. A table of the cubes of numbers can be produced using this new procedure as follows:

```
?- cube.
2.
8
5.
125
12.
1728
stop.
yes
```

The numbers 2, 5 and 12 were typed in by the user on the terminal; the other numbers were output by the program. Note that each number entered by the user had to be followed by a full stop, which signals the end of a term.

It may appear that the above **cube** procedure could be simplified. However, the following attempt to simplify is not correct:

```
cube :-
  read( stop), !.
```

```
cube :-
  read( N),
  C is N * N * N,
  write( C),
  cube.
```

The reason why this is wrong can be seen easily if we trace the program with input data 5, say. The goal **read(stop)** will fail when the number is read, and this number will be lost for ever. The next **read** goal will input the next term. On the other hand, it could happen that the **stop** signal is read by the goal **read(N)**, which would then cause a request to multiply non-numeric data.

The **cube** procedure conducts interaction between the user and the program. In such cases it is usually desirable that the program, before reading new data from the terminal, signals to the user that it is ready to accept the information, and perhaps also says what kind of information it is expecting. This is usually done by sending a 'prompt' signal to the user before reading. Our **cube** procedure would be accordingly modified, for example, as follows:

```
cube :-
  write( 'Next item, please: '),
  read( X),
  process( X).

process( stop) :- !.

process( N) :-
  C is N * N * N,
  write( 'Cube of '), write( N), write( 'is '),
  write( C), nl,
  cube.
```

A conversation with this new version of **cube** would then be, for example, as follows:

```
?- cube.
Next item, please: 5.
Cube of 5 is 125
Next item, please: 12.
Cube of 12 is 1728
Next item, please: stop.
yes
```

Depending on the implementation, an extra request (like **ttyflush**, say) after writing the prompt might be necessary in order to force the prompt to actually appear on the screen before reading.

In the following sections we will look at some typical examples of operations that involve reading and writing.

6.2.2 Displaying lists

Besides the standard Prolog format for lists, there are several other natural forms for displaying lists which have advantages in some situations. The following procedure

```
writelist( L)
```

outputs a list L so that each element of L is written on a separate line:

```
writelist( [] ).

writelist( [X | L] ) :-
  write( X), nl,
  writelist( L).
```

If we have a list of lists, one natural output form is to write the elements of each list in one line. To this end, we will define the procedure **writelist2**. An example of its use is:

```
?- writelist2( [ [a,b,c], [d,e,f], [g,h,i] ] ).

a b c
d e f
g h i
```

A procedure that accomplishes this is:

```
writelist2( [] ).

writelist2( [L | LL] ) :-
  doline( L), nl,
  writelist2( LL).

doline( [] ).

doline( [X | L] ) :-
  write( X), tab( 1),
  doline( L).
```

A list of integer numbers can be sometimes conveniently shown as a bar graph. The following procedure, **bars**, will display a list in this form, assuming that the numbers in the list are between 0 and 80. An example of using **bars** is:

```
?- bars( [3,4,6,5] ).
***
****
******
*****
```

The **bars** procedure can be defined as follows:

```
bars( [N | L] ) :-
  stars( N), nl,
  bars( L).

stars( N) :-
  N > 0,
  write( *),
  N1 is N - 1,
  stars( N1).

stars( N) :-
  N =< 0.
```

6.2.3 Formatting terms

Let us suppose that our program deals with families that are represented as terms, as in Chapter 4 (Figure 4.1). Then, for example, if F is instantiated to the term shown in Figure 4.1, the goal

```
write( F)
```

will cause this term to be output in the standard form, something like this:

```
family(person(tom,fox,date(7,may,1950),works(bbc,15200)),
person(ann,fox,date(9,may,1951),unemployed),[person(pat,
fox,date(5,may,1973),unemployed),person(jim,fox,date(5,
may,1973),unemployed)])
```

This contains all the information, but the form is rather confusing as it is hard to follow what parts of information form semantic units. We would therefore

---

parents

tom fox, born 7 may 1950, works bbc, salary 15200
ann fox, born 9 may 1951, unemployed

children

pat fox, born 5 may 1973, unemployed
jim fox, born 5 may 1973, unemployed

Figure 6.2 Improved format for family terms.

normally prefer to have this displayed in a formatted manner; for example, as shown in Figure 6.2. The procedure, shown in Figure 6.3,

writefamily(F)

achieves this format.

6.2.4 Processing a file of terms

A typical sequence of goals to process a whole file, F, would look something like this:

..., **see(F), processfile, see(user)**, ...

Here **processfile** is a procedure to read and process each term in F, one after

```
writefamily( family( Husband, Wife, Children) ) :-
  nl, write( parents), nl, nl,
  writeperson( Husband), nl,
  writeperson( Wife), nl, nl,
  write( children), nl, nl,
  writepersonlist( Children).

writeperson( person( Firstname, Secname, date(D,M,Y), Work) ) :-
  tab( 4), write( Firstname),
  tab( 1), write( Secname),
  write( ', born'),
  write( D), tab( 1),
  write( M), tab( 1),
  write( Y), write( ','),
writework( Work).

writepersonlist( [] ).

writepersonlist( [P | L] ) :-
  writeperson( P), nl,
  writepersonlist( L).

writework( unemployed) :-
  write( unemployed).

writework( works( Comp, Sal) ) :-
  write( 'works'), write( Comp),
  write( ', salary'), write( Sal).
```

Figure 6.3 A program to produce the format of Figure 6.2.

another, until the end of the file is encountered. A typical schema for **processfile** is:

```
processfile :-
  read( Term),
  process( Term).

process( end_of_file) :- !.          % All done

process( Term) :-
  treat( Term),                      % Process current item
  processfile.                       % Process rest of file
```

Here **treat(Term)** represents whatever is to be done with each term. An example would be a procedure to display on the terminal each term together with its consecutive number. Let us call this procedure **showfile**. It has to have an additional argument to count the terms read:

```
showfile( N) :-
  read( Term),
  show( Term, N).

show( end_of_file, _) :- !.

show( Term, N) :-
  write( N), tab( 2), write( Term),
  N1 is N + 1,
  showfile( N1).
```

Another example of using this schema for processing a file is as follows. We have a file, named **file1**, of terms of the form:

```
item( ItemNumber, Description, Price, SupplierName)
```

Each term describes an entry in a catalogue of items. We want to produce another file that contains only items supplied by a specified supplier. As the supplier, in this new file, will always be the same, his or her name need only be written at the beginning of the file, and omitted from other terms. The procedure will be:

```
makefile( Supplier)
```

For example, if the original catalogue is stored in **file1**, and we want to produce the special catalogue on **file2** of everything that Harrison supplies, then we would use the **makefile** procedure as follows:

```
?- see( file1), tell( file2), makefile( harrison), see( user), tell( user).
```

The procedure **makefile** can be defined as follows:

```
makefile( Supplier) :-
  write( Supplier), write( '.' ), nl,
  makerest( Supplier).

makerest( Supplier) :-
  read( Item),
  process( Item, Supplier).

process( end_of_file, _) :- !.

process( item( Num, Desc, Price, Supplier), Supplier) :- !,
  write( item( Num, Desc, Price) ),
  write( '.' ), nl,
  makerest( Supplier).

process( _, Supplier) :-
  makerest( Supplier).
```

Notice that **process** writes full stops between terms to make future reading of the file possible by the **read** procedure.

Exercises

6.1 Let **f** be a file of terms. Define a procedure

findterm(Term)

that displays on the terminal the first term in **f** that matches **Term**.

6.2 Let **f** be a file of terms. Write a procedure

findallterms(Term)

that displays on the terminal all the terms in **f** that match **Term**. Make sure that **Term** is not instantiated in the process (which could prevent its match with terms that occur later in the file).

6.3 Manipulating characters

A character is written on the current output stream with the goal

put(C)

where C is the ASCII code (a number between 0 and 127) of the character to be output. For example, the question

```
?- put( 65), put( 66), put( 67).
```

would cause the following output:

ABC

65 is the ASCII code of 'A', 66 of 'B', 67 of 'C'.

A single character can be read from the current input stream by the goal

```
get0( C)
```

This causes the current character to be read from the input stream, and the variable C becomes instantiated to the ASCII code of this character. A variation of the predicate **get0** is **get**, which is used for reading non-blank characters. So the goal

```
get( C)
```

will cause the skipping over of all non-printable characters (blanks in particular) from the current input position in the input stream up to the first printable character. This character is then also read and C is instantiated to its ASCII code.

As an example of using predicates that transfer single characters let us define a procedure, **squeeze**, to do the following: read a sentence from the current input stream, and output the same sentence reformatted so that multiple blanks between words are replaced by single blanks. For simplicity we will assume that any input sentence processed by **squeeze** ends with a full stop and that words are separated simply by one or more blanks, but no other character. An acceptable input is then:

```
The      robot tried     to pour wine out     of the      bottle.
```

The goal **squeeze** would output this in the form:

```
The robot tried to pour wine out of the bottle.
```

The **squeeze** procedure will have a similar structure to the procedures for processing files in the previous section. First it will read the first character, output this character, and then complete the processing depending on this character. There are three alternatives that correspond to the following cases: the character is either a full stop, a blank or a letter. The mutual exclusion of the three alternatives is achieved in the program by cuts:

```
squeeze :-
  get0( C),
  put( C),
  dorest( C).

dorest( 46) :- !.              % 46 is ASCII for full stop, all done
```

```
dorest( 32) :- !,          % 32 is ASCII for blank
  get( C),                 % Skip other blanks
  put( C),
  dorest( C).

dorest( Letter) :-
  squeeze.
```

Exercise

6.3 Generalize the **squeeze** procedure to handle commas as well. All blanks immediately preceding a comma are to be removed, and we want to have one blank after each comma.

6.4 Constructing and decomposing atoms

It is often desirable to have information, read as a sequence of characters, represented in the program as an atom. There is a built-in predicate, **name**, which can be used to this end. **name** relates atoms and their ASCII encodings. Thus

```
name( A, L)
```

is true if L is the list of ASCII codes of the characters in A. For example

```
name( zx232, [122,120,50,51,50] )
```

is true. There are two typical uses of **name**:

(1) given an atom, break it down into single characters;
(2) given a list of characters, combine them into an atom.

An example of the first kind of application would be a program that deals with orders, taxies and drivers. These would be, in the program, represented by atoms such as:

```
order1, order2, driver1, driver2, taxia1, taxilux
```

The following predicate

```
taxi( X)
```

tests whether an atom X represents a taxi:

```
taxi( X) :-
  name( X, Xlist),
  name( taxi, Tlist),
  conc( Tlist, _, Xlist).          % Is word 'taxi' prefix of X?

conc( [], L, L).

conc( [A | L1], L2, [A | L3] ) :-
  conc( L1, L2, L3).
```

Predicates **order** and **driver** can be defined analogously.

The next example illustrates the use of combining characters into atoms. We will define a predicate

getsentence(Wordlist)

that reads a free-form natural language sentence and instantiates **Wordlist** to some internal representation of the sentence. A natural choice for the internal representation, which would enable further processing of the sentence, is this: each word of the input sentence is represented as a Prolog atom; the whole sentence is represented as a list of atoms. For example, if the current input stream is

Mary was pleased to see the robot fail.

then the goal **getsentence(Sentence)** will cause the instantiation

Sentence = ['Mary', was, pleased, to, see, the, robot, fail]

For simplicity, we will assume that each sentence terminates with a full stop and that there are no punctuation symbols within the sentence.

The program is shown in Figure 6.4. The procedure **getsentence** first reads the current input character, **Char**, and then supplies this character to the procedure **getrest** to complete the job. **getrest** has to react properly according to three cases:

(1) **Char** is the full stop: then everything has been read.

(2) **Char** is the blank: ignore it, **getsentence** from rest of input.

(3) **Char** is a letter: first read the word, **Word**, which begins with **Char**, and then use **getsentence** to read the rest of the sentence, producing **Wordlist**. The cumulative result is the list **[Word | Wordlist]**.

The procedure that reads the characters of one word is

getletters(Letter, Letters, Nextchar)

```
/*
  Procedure getsentence reads in a sentence and combines the
  words into a list of atoms. For example

     getsentence( Wordlist)

  produces

     Wordlist = [ 'Mary', was, pleased, to, see, the, robot, fail]

  if the input sentence is:

     Mary was pleased to see the robot fail.
*/

getsentence( Wordlist) :-
  get0( Char),
  getrest( Char, Wordlist).

getrest( 46, [] ) :- !.                    % End of sentence: 46 = ASCII for '.'

getrest( 32, Wordlist) :- !,               % 32 = ASCII for blank
  getsentence( Wordlist).                  % Skip the blank

getrest( Letter, [Word | Wordlist] ) :-
  getletters( Letter, Letters, Nextchar),  % Read letters of current word
  name( Word, Letters),
  getrest( Nextchar, Wordlist).

getletters( 46, [], 46) :- !.              % End of word: 46 = full stop

getletters( 32, [], 32) :- !.              % End of word: 32 = blank

getletters( Let, [Let | Letters], Nextchar) :-
  get0( Char),
  getletters( Char, Letters, Nextchar).
```

Figure 6.4 A procedure to transform a sentence into a list of atoms.

The three arguments are:

(1) **Letter** is the current letter (already read) of the word being read.
(2) **Letters** is the list of letters (starting with **Letter**) up to the end of the word.
(3) **Nextchar** is the input character that immediately follows the word read. **Nextchar** must be a non-letter character.

We conclude this example with a comment on the possible use of the **getsentence** procedure. It can be used in a program to process text in natural

language. Sentences represented as lists of words are in a form that is suitable for further processing in Prolog. A simple example is to look for certain keywords in input sentences. A much more difficult task would be to understand the sentence; that is, to extract from the sentence its meaning, represented in some chosen formalism. This is an important research area of Artificial Intelligence.

Exercises

6.4 Define the relation

starts(Atom, Character)

to check whether **Atom** starts with **Character**.

6.5 Define the procedure **plural** that will convert nouns into their plural form. For example:

```
?- plural( table, X).

X = tables
```

6.6 Write the procedure

search(KeyWord, Sentence)

that will, each time it is called, find a sentence in the current input file that contains the given **KeyWord**. **Sentence** should be in its original form, represented as a sequence of characters or as an atom (procedure **getsentence** of this section can be accordingly modified).

6.5 Reading programs: *consult, reconsult*

We can communicate our programs to the Prolog system by means of two built-in predicates: **consult** and **reconsult**. We tell Prolog to read a program from a file F with the goal:

```
?- consult( F).
```

The effect will be that all clauses in F are read and will be used by Prolog when answering further questions from the user. If another file is 'consulted' at some later time during the same session, clauses from this new file are simply added at the end of the current set of clauses.

We do not have to enter our program into a file and then request 'consulting' that file. Instead of reading a file, Prolog can also accept our

program directly from the terminal, which corresponds to the pseudo-file **user**. We can achieve this by:

```
?- consult( user).
```

Now Prolog is waiting for program clauses to be entered from the terminal.

A shorthand notation for consulting files is available in some Prolog systems. Files that are to be consulted are simply put into a list and stated as a goal. For example:

```
?- [ file1, file2, file3].
```

This is exactly equivalent to three goals:

```
?- consult( file1), consult( file2), consult( file3).
```

The built-in predicate **reconsult** is similar to **consult**. A goal

```
?- reconsult( F).
```

will have the same effect as **consult(F)** with one exception. If there are clauses in F about a relation that has been previously defined, the old definition will be superseded by the new clauses about this relation in F. The difference between **consult** and **reconsult** is that **consult** always *adds* new clauses while **reconsult** *redefines* previously defined relations. **reconsult(F)** will, however, not affect any relation about which there is no clause in F.

It should be noted, again, that the details of 'consulting' files depend on the implementation of Prolog, as is the case with most other built-in procedures.

Summary

- Input and output (other than that associated with querying the program) is done using built-in procedures. This chapter introduced a simple and practical repertoire of such procedures that can be found in many Prolog implementations.
- Files are sequential. There is the *current input stream* and the *current output stream*. The user terminal is treated as a file called **user**.
- Switching between streams is done by:

see(File)	**File** becomes the current input stream
tell(File)	**File** becomes the current output stream
seen	close the current input stream
told	close the current output stream

- Files are read and written in two ways:

  as sequences of characters
  as sequences of terms

  Built-in procedures for reading and writing characters and terms are:

read(Term)	input next term
write(Term)	output **Term**
put(CharCode)	output character with the given ASCII code
get0(CharCode)	input next character
get(CharCode)	input next 'printable' character

- Two procedures help formatting:

nl	output new line
tab(N)	output N blanks

- The procedure **name(Atom, CodeList)** decomposes and constructs atoms. **CodeList** is the list of ASCII codes of the characters in **Atom**.

7 More Built-in Procedures

In this chapter we will examine some more built-in procedures for advanced Prolog programming. These features enable the programming of operations that are not possible using only the features introduced so far. One set of such procedures manipulates terms: testing whether some variable has been instantiated to an integer, taking terms apart, constructing new terms, etc. Another useful set of procedures manipulates the 'database': they add new relations to the program or remove existing ones.

The built-in procedures largely depend on the implementation of Prolog. However, the procedures discussed in this chapter are provided by many Prolog implementations. Various implementations may provide additional features.

7.1 Testing the type of terms

7.1.1 Predicates *var, nonvar, atom, integer, atomic*

Terms may be of different types: variable, integer, atom, etc. If a term is a variable then it can be, at some point during the execution of the program, instantiated or uninstantiated. Further, if it is instantiated, its value can be an atom, a structure, etc. It is sometimes useful to know what is the type of this value. For example, we may want to add the values of two variables, X and Y, by

```
Z is X + Y
```

Before this goal is executed, X and Y have to be instantiated to integers. If we are not sure that X and Y will indeed be instantiated to integers then we should check this in the program before arithmetic is done.

To this end we can use the built-in predicate **integer**. **integer(X)** is true if X is an integer or if it is a variable whose value is an integer. We say that X must 'currently stand for' an integer. The goal of adding X and Y can then be protected by the following test on X and Y:

```
..., integer( X), integer( Y), Z is X + Y, ...
```

If X and Y are not both integers then no arithmetic will be attempted. So the **integer** goals 'guard' the goal **Z is X + Y** before meaningless execution.

Built-in predicates of this sort are: **var, nonvar, atom, integer, atomic.** Their meaning is as follows:

var(X)

This goal succeeds if X is currently an uninstantiated variable.

nonvar(X)

This goal succeeds if X is a term other than a variable, or X is an already instantiated variable.

atom(X)

This is true if X currently stands for an atom.

integer(X)

This goal is true if X currently stands for an integer.

atomic(X)

This goal is true if X currently stands for an integer or an atom.

The following example questions to Prolog illustrate the use of these built-in predicates:

```
?- var( Z), Z = 2.
Z = 2

?- Z = 2, var( Z).
no

?- integer( Z), Z = 2.
no

?- Z = 2, integer( Z), nonvar( Z).
Z = 2

?- atom( 22).
no

?- atomic( 22).
yes
```

```
?- atom( ==> ).

yes

?- atom( p(1) ).

no
```

We will illustrate the need for **atom** by an example. We would like to count how many times a given atom occurs in a given list of objects. To this purpose we will define a procedure

```
count( A, L, N)
```

where A is the atom, L is the list and N is the number of occurrences. The first attempt to define **count** could be:

```
count( _, [], 0).

count( A, [A | L], N) :- !,
  count( A, L, N1),                % N1 = number of occurrences in tail
  N is N1 + 1.

count( A, [_ | L], N) :-
  count( A, L, N).
```

Now let us try to use this procedure on some examples:

```
?- count( a, [a,b,a,a], N).

N = 3

?- count( a, [a,b,X,Y], Na).

Na = 3

...

?- count( b, [a,b,X,Y], Nb).

Nb = 3

...

?- L = [a, b, X, Y], count( a, L, Na), count( b, L, Nb).

Na = 3
Nb = 1
X = a
Y = a
...
```

In the last example, X and Y both became instantiated to **a** and therefore we only got Nb = 1; but this is not what we had in mind. We are interested in the number of real occurrences of the given *atom*, and not in the number of terms that *match* this atom. According to this more precise definition of the **count** relation we have to check whether the head of the list is an atom. The modified program is as follows:

```
count( _, [], 0).

count( A, [B | L], N) :-
  atom( B), A = B, !,          % B is atom A?
    count( A, L, N1),          % Count in tail
    N is N1 + 1;
  count( A, L, N).             % Otherwise just count the tail
```

The following, more complex programming exercise in solving cryptarithmetic puzzles makes use of the **nonvar** predicate.

7.1.2 A cryptarithmetic puzzle using *nonvar*

A popular example of a cryptarithmetic puzzle is

```
   DONALD
+  GERALD
   ------
   ROBERT
```

The problem here is to assign decimal digits to the letters D, O, N, etc., so that the above sum is valid. All letters have to be assigned different digits, otherwise trivial solutions are possible – for example, all letters equal zero.

We will define a relation

```
sum( N1, N2, N)
```

where N1, N2 and N represent the three numbers of a given cryptarithmetic puzzle. The goal **sum(N1, N2, N3)** is true if there is an assignment of digits to letters such that N1 + N2 = N.

The first step toward finding a solution is to decide how to represent the numbers N1, N2 and N in the program. One way of doing this is to represent each number as a list of decimal digits. For example, the number 225 would be represented by the list **[2,2,5]**. As these digits are not known in advance, an uninstantiated variable will stand for each digit. Using this representation, the problem can be depicted as:

```
  [D,O,N,A,L,D]
+ [G,E,R,A,L,D]
= [R,O,B,E,R,T]
```

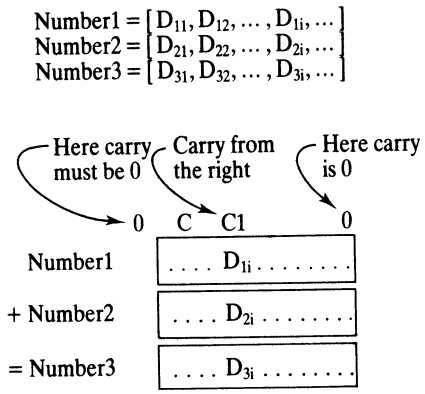


Figure 7.1 Digit-by-digit summation. The relations at the indicated ith digit position are: $D_{3i} = (C1 + D_{1i} + D_{2i}) \bmod 10$; $C = (C1 + D_{1i} + D_{2i}) \text{ div } 10$.

The task is to find such an instantiation of the variables D, O, N, etc., for which the sum is valid. When the **sum** relation has been programmed, the puzzle can be stated to Prolog by the question:

```
?- sum( [D,O,N,A,L,D], [G,E,R,A,L,D], [R,O,B,E,R,T] ).
```

To define the **sum** relation on lists of digits, we have to implement the actual rules for doing summation in the decimal number system. The summation is done digit by digit, starting with the right-most digits, continuing toward the left, always taking into account the carry digit from the right. It is also necessary to maintain a set of available digits; that is, digits that have not yet been used for instantiating variables already encountered. So, in general, besides the three numbers N1, N2 and N, some additional information is involved, as illustrated in Figure 7.1:

- carry digit before the summation of the numbers
- carry digit after the summation
- set of digits available before the summation
- remaining digits, not used in the summation

To formulate the **sum** relation we will use, once again, the principle of generalization of the problem: we will introduce an auxiliary, more general relation, **sum1**. **sum1** has some extra arguments, which correspond to the above additional information:

```
sum1( N1, N2, N, C1, C, Digits1, Digits)
```

N1, N2 and N are our three numbers, as in the **sum** relation, C1 is carry from

the right (before summation of N1 and N2), and C is carry to the left (after the summation). The following example illustrates:

```
?- sum1( [H,E], [6,E], [U,S], 1, 1, [1,3,4,7,8,9], Digits).

H = 8
E = 3
S = 7
U = 4
Digits = [1,9]
```

As Figure 7.1 shows, C1 and C have to be 0 if N1, N2 and N are to satisfy the **sum** relation. **Digits1** is the list of available digits for instantiating the variables in N1, N2 and N; **Digits** is the list of digits that were not used in the instantiation of these variables. Since we allow the use of any decimal digit in satisfying the **sum** relation, the definition of **sum** in terms of **sum1** is as follows:

```
sum( N1, N2, N) :-
  sum1( N1, N2, N, 0, 0, [0,1,2,3,4,5,6,7,8,9], _).
```

The burden of the problem has now shifted to the **sum1** relation. This relation is, however, general enough so that it can be defined recursively. We will assume, without loss of generality, that the three lists representing the three numbers are of equal length. Our example problem, of course, satisfies this constraint; if not a 'shorter' number can be prefixed by zeros.

The definition of **sum1** can be divided into two cases:

(1) The three numbers are represented by empty lists. Then:

```
sum1( [], [], [], 0, 0, Digs, Digs).
```

(2) All three numbers have some left-most digit and the remaining digits on their right. So they are of the form:

```
[D1 | N1],      [D2 | N2],      [D | N]
```

In this case two conditions must be satisfied:

(a) The remaining digits themselves, viewed as three numbers N1, N2 and N, have to satisfy the **sum1** relation, giving some carry digit, C2, to the left, and leaving some unused subset of decimal digits, **Digs2**.

(b) The left-most digits D1, D2 and D, and the carry digit C2 have to satisfy the relation indicated in Figure 7.1: C2, D1 and D2 are added giving D and a carry to the left. This condition will be formulated in our program as a relation **digitsum**.

Translating this case into Prolog we have:

```
sum1( [D1 | N1], [D2 | N2], [D | N], C1, C, Digs1, Digs) :-
  sum1( N1, N2, N, C1, C2, Digs1, Digs2),
  digitsum( D1, D2, C2, D, C, Digs2, Digs).
```

It only remains to define the **digitsum** relation in Prolog. There is one subtle detail that involves the use of the metalogical predicate **nonvar**. D1, D2 and D have to be decimal digits. If any of them is not yet instantiated then it has to become instantiated to one of the digits in the list **Digs2**. Once it is instantiated to one of the digits, this digit has to be deleted from the set of available digits. If D1, D2 or D is already instantiated then, of course, none of the available digits will be spent. This is realized in the program as a non-deterministic deletion of an item from a list. If this item is non-variable then nothing

```
% Solving cryptarithmetic puzzles

sum( N1, N2, N) :-                    % Numbers represented as lists of digits
  sum1( N1, N2, N,
        0, 0,                         % Carries from right and to left both 0
        [0,1,2,3,4,5,6,7,8,9], _).    % All digits available

sum1( [], [], [], 0, 0, Digits, Digits).

sum1( [D1 | N1], [D2 | N2], [D | N], C1, C, Digs1, Digs) :-
  sum1( N1, N2, N, C1, C2, Digs1, Digs2),
  digitsum( D1, D2, C2, D, C, Digs2, Digs).

digitsum( D1, D2, C1, D, C, Digs1, Digs) :-
  del( D1, Digs1, Digs2),             % Select an available digit for D1
  del( D2, Digs2, Digs3),             % Select an available digit for D2
  del( D, Digs3, Digs),               % Select an available digit for D
  S is D1 + D2 + C1,
  D is S mod 10,
  C is S div 10.

del( A, L, L) :-
  nonvar( A), !.                      % A already instantiated

del( A, [A | L], L).

del( A, [B | L], [B | L1] ) :-
  del( A, L, L1).

% Some puzzles

puzzle1( [D,O,N,A,L,D],
         [G,E,R,A,L,D],
         [R,O,B,E,R,T] ).

puzzle2( [0,S,E,N,D],
         [0,M,O,R,E],
         [M,O,N,E,Y] ).
```

Figure 7.2 A program for cryptarithmetic puzzles.

is deleted (no instantiation occurs). This is programmed as:

```
del( Item, List, List) :-
  nonvar( Item), !.

del( Item, [Item | List], List).

del( Item, [A | List], [A | List1] ) :-
  del( Item, List, List1).
```

A complete program for cryptarithmetic puzzles is shown in Figure 7.2. The program also includes the definition of two puzzles. The question to Prolog about DONALD, GERALD and ROBERT, using this program, would be:

```
?- puzzle1( N1, N2, N), sum( N1, N2, N).
```

Sometimes this puzzle is made easier by providing part of the solution as an additional constraint that D be equal 5. The puzzle in this form could be communicated to Prolog using **sum1**:

```
?- sum1( [5,O,N,A,L,5],
         [G,E,R,A,L,5],
         [R,O,B,E,R,T],
         0, 0, [0,1,2,3,4,6,7,8,9], _).
```

It is interesting that in both cases there is only one solution. That is, there is only one way of assigning digits to letters.

Exercises

7.1 Write a procedure **simplify** to symbolically simplify summation expressions with numbers and symbols (lower-case letters). Let the procedure rearrange the expressions so that all the symbols precede numbers. These are examples of its use:

```
?- simplify( 1 + 1 + a, E).
E = a + 2

?- simplify( 1 + a + 4 + 2 + b + c, E).
E = a + b + c + 7

?- simplify( 3 + x + x, E).
E = 2*x + 3
```

7.2 Define the procedure

```
add( Item, List)
```

to store a new element into a list. Assume that all of the elements that can be stored are atoms. **List** contains all the stored elements followed by a

tail that is not instantiated and can thus accommodate new elements. For example, let the existing elements stored be **a**, **b** and **c**. Then

List = [a, b, c | Tail]

where **Tail** is a variable. The goal

add(d, List)

will cause the instantiation

Tail = [d | NewTail] and **List = [a, b, c, d | NewTail]**

Thus the structure can, in effect, grow by accepting new items. Define also the corresponding membership relation.

7.2 Constructing and decomposing terms: *=.., functor, arg, name*

There are three built-in predicates for decomposing terms and constructing new terms: **functor**, **arg** and =... We will first look at =.., which is written as an infix operator. The goal

Term =.. L

is true if L is a list that contains the principal functor of **Term**, followed by its arguments. The following examples illustrate:

```
?- f( a, b) =.. L.
L = [f, a, b]
?- T =.. [rectangle, 3, 5].
T = rectangle( 3, 5)
?- Z =.. [p, X, f(X,Y)]
Z = p( X, f(X,Y) )
```

Why would we want to decompose a term into its components – its functor and its arguments? Why construct a new term from a given functor and arguments? The following example illustrates the need for this.

Let us consider a program that manipulates geometric figures. Figures are squares, rectangles, triangles, circles, etc. They can, in the program, be represented as terms such that the functor indicates the type of figure, and the arguments specify the size of the figure, as follows:

```
square( Side)
triangle( Side1, Side2, Side3)
circle( R)
```

One operation on such figures can be enlargement. We can implement this as a three-argument relation

enlarge(Fig, Factor, Fig1)

where **Fig** and **Fig1** are geometric figures of the same type (same functor), and the parameters of **Fig1** are those of **Fig** multiplicatively enlarged by **Factor**. For simplicity, we will assume that all the parameters of **Fig** are already known; that is, instantiated to numbers, and so is **Factor**. One way of programming the **enlarge** relation is:

```
enlarge( square(A), F, square(A1) ) :-
  A1 is F*A.

enlarge( circle(R), F, circle(R1) ) :-
  R1 is F*R1.

enlarge( rectangle(A,B), F, rectangle(A1,B1) ) :-
  A1 is F*A, B1 is F*B.

...
```

This works, but it is awkward when there are many different figure types. We have to foresee all types that may possibly occur. Thus, we need an extra clause for each type although each clause says essentially the same thing: take the parameters of the original figure, multiply all the parameters by the factor, and make a figure of the same type with new parameters.

One (unsuccessful) attempt to handle, at least, all one-parameter figures with one clause could be:

```
enlarge( Type(Par), F, Type(Par1) ) :-
  Par1 is F*Par.
```

However, this is normally not allowed in Prolog because the functor has to be an atom; so the variable **Type** would not be accepted syntactically as a functor. The correct method is to use the predicate '=..'. Then the **enlarge** procedure can be stated completely generally, for any type of object, as follows:

```
enlarge( Fig, F, Fig1) :-
  Fig =.. [Type | Parameters],
  multiplylist( Parameters, F, Parameters1),
  Fig1 =.. [Type | Parameters1].

multiplylist( [], _, [] ).

multiplylist( [X | L], F, [X1 | L1] ) :-
  X1 is F*X, multiplylist( L, F, L1).
```

Our next example of using the '=..' predicate comes from symbolic manipulation of formulas where a frequent operation is to substitute some

subexpression by another expression. We will define the relation

substitute(Subterm, Term, Subterm1, Term1)

as follows: if all occurrences of **Subterm** in **Term** are substituted by **Subterm1** then we get **Term1**. For example:

```
?- substitute( sin(x), 2*sin(x)*f(sin(x)), t, F).

F = 2*t*f(t)
```

By 'occurrence' of **Subterm** in **Term** we will mean something in **Term** that *matches* **Subterm**. We will look for occurrences from top to bottom. So the goal

```
?- substitute( a+b, f( a, A+B), v, F).
```

will produce

```
F = f( a, v)          and not     F = f( a, v+v)
A = a                             A = a+b
B = b                             B = a+b
```

In defining the **substitute** relation we have to consider the following decisions depending on the case:

If **Subterm** = **Term** then **Term1** = **Subterm1**;
otherwise if **Term** is 'atomic' (not a structure)
  then **Term1** = **Term** (nothing to be substituted),
  otherwise the substitution is to be carried out on the arguments of **Term**.

These rules can be converted into a Prolog program, shown in Figure 7.3.

Terms that are constructed by the '=..' predicate can be, of course, also used as goals. The advantage of this is that the program itself can, during execution, generate and execute goals of forms that were not necessarily foreseen at the time of writing the program. A sequence of goals illustrating this effect would be something like the following:

```
obtain( Functor),
compute( Arglist),
Goal =.. [Functor | Arglist],
Goal
```

Here, **obtain** and **compute** are some user-defined procedures for getting the components of the goal to be constructed. The goal is then constructed by '=..', and invoked for execution by simply stating its name, **Goal**.

```
% Relation
%
% substitute( Subterm, Term, Subterm1, Term1)
%
% is: if all occurrences of Subterm in Term are substituted
% with Subterm1 when we get Term1.

% Case 1: Substitute whole term

substitute( Term, Term, Term1, Term1) :- !.

% Case 2: Nothing to substitute

substitute( _, Term, _, Term) :-
  atomic( Term), !.

% Case 3: Do substitution on arguments

substitute( Sub, Term, Sub1, Term1) :-
  Term =.. [F | Args],                    % Get arguments
  substlist( Sub, Args, Sub1, Args1),     % Perform substitution on them
  Term1 =.. [F | Args1].

substlist( _, [], _, [] ).

substlist( Sub, [Term | Terms], Sub1, [Term1 | Terms1] ) :-
  substitute( Sub, Term, Sub1, Term1),
  substlist( Sub, Terms, Sub1, Terms1).
```

Figure 7.3 A procedure for substituting a subterm of a term by another subterm.

Some implementations of Prolog may require that all the goals, as they appear in the program, are *syntactically* either atoms or structures with an atom as the principal functor. Thus a variable, regardless of its eventual instantiation, in such a case may not be syntactically acceptable as a goal. This problem is circumvented by another built-in predicate, **call**, whose argument is the goal to be executed. Accordingly, the example would be rewritten as:

```
...
Goal =.. [Functor | Arglist],
call( Goal)
```

Sometimes we may want to extract from a term just its principal functor or one of its arguments. In such a case we can, of course, use the '=..' relation. But it can be neater and more practical, and also more efficient, to use one of the other two built-in procedures for manipulating terms: **functor** and **arg**. Their meaning is as follows: a goal

```
functor( Term, F, N)
```

is true if F is the principal functor of **Term** and **N** is the arity of F. A goal

```
arg( N, Term, A)
```

is true if A is the Nth argument in **Term**, assuming that arguments are numbered from left to right starting with 1. The following examples illustrate:

```
?- functor( t( f(X), X, t), Fun, Arity).

Fun = t
Arity = 3

?- arg( 2, f( X, t(a), t(b)), Y).

Y = t(a)

?- functor( D, date, 3),
   arg( 1, D, 29),
   arg( 2, D, june),
   arg( 3, D, 1982).

D = date( 29, june, 1982)
```

The last example shows a special application of the **functor** predicate. The goal **functor(D, date, 3)** generates a 'general' term whose principal functor is **date** with three arguments. The term is general in that the three arguments are uninstantiated variables whose names are generated by Prolog. For example:

```
D = date( _5, _6, _7)
```

These three variables are then instantiated in the example above by the three **arg** goals.

Related to this set of built-in predicates is the predicate **name** for constructing/decomposing atoms, introduced in Chapter 6. We will repeat its meaning here for completeness.

```
name( A, L)
```

is true if L is the list of ASCII codes of the characters in atom A.

Exercises

7.3 Define the predicate **ground(Term)** so that it is true if **Term** does not contain any uninstantiated variables.

7.4 The **substitute** procedure of this section only produces the 'outer-most' substitution when there are alternatives. Modify the procedure so that all possible alternative substitutions are produced through backtracking.

For example:

```
?- substitute( a+b, f(A+B), new, NewTerm).
A = a
B = b
NewTerm = f( new);
A = a+b
B = a+b
NewTerm = f( new+new)
```

Our original version only finds the first answer.

7.5 Define the relation

subsumes(Term1, Term2)

so that **Term1** is more general than **Term2**. For example:

```
?- subsumes( X, c).
yes
?- subsumes( g(X), g(t(Y)) ).
yes
?- subsumes( f(X,X), f(a,b) ).
no
```

7.3 Various kinds of equality

When do we consider two terms to be equal? Until now we have introduced three kinds of equality in Prolog. The first was based on matching, written as:

X = Y

This is true if X and Y match. Another type of equality was written as

X is E

This is true if X matches the value of the arithmetic expression E. We also had:

E1 =:= E2

This is true if the values of the arithmetic expressions E1 and E2 are equal. In contrast, when the values of two arithmetic expressions are not equal, we write

E1 =\= E2

Sometimes we are interested in a stricter kind of equality: the *literal equality* of two terms. This kind of equality is implemented as another built-in

predicate written as an infix operator '==':

```
T1 == T2
```

This is true if terms T1 and T2 are identical; that is, they have exactly the same structure and all the corresponding components are the same. In particular, the names of the variables also have to be the same. The complementary relation is 'not identical', written as:

```
T1 \== T2
```

Here are some examples:

```
?- f( a, b) == f( a, b).
yes
?- f( a, b) == f( a, X).
no
?- f( a, X) == f( a, Y).
no
?- X \== Y.
yes
?- t( X, f(a,Y) ) == t( X, f(a,Y) ).
yes
```

As an example, let us redefine the relation

```
count( Term, List, N)
```

from Section 7.1. This time let N be the number of literal occurrences of the term **Term** in a list **List**:

```
count( _, [], 0).
count( Term, [Head | L ], N) :-
  Term == Head, !,
    count( Term, L, N1),
    N is N1 + 1;
  count( Term, L, N).
```

7.4 Database manipulation

According to the relational model of databases, a database is a specification of a set of relations. A Prolog program can be viewed as such a database: the specification of relations is partly explicit (facts) and partly implicit (rules).

Furthermore, built-in predicates make it possible to update this database during the execution of the program. This is done by adding (during execution) new clauses to the program or by deleting existing clauses. Predicates that serve these purposes are **assert**, **asserta**, **assertz** and **retract**.

A goal

```
assert( C)
```

always succeeds and, as its side effect, causes a clause C to be 'asserted' – that is, added to the database. A goal

```
retract( C)
```

does the opposite: it deletes a clause that matches C. The following conversation with Prolog illustrates:

```
?- crisis.

no

?- assert( crisis).

yes

?- crisis.

yes

?- retract( crisis).

yes

?- crisis.

no
```

Clauses thus asserted act exactly as part of the 'original' program. The following example shows the use of **assert** and **retract** as one method of handling changing situations. Let us assume that we have the following program about weather:

```
nice :-
  sunshine, not raining.

funny :-
  sunshine, raining.

disgusting :-
  raining, fog.

raining.

fog.
```

The following conversation with this program will gradually update the database:

```
?- nice.

no

?- disgusting.

yes

?- retract( fog).

yes

?- disgusting.

no

?- assert( sunshine).

yes

?- funny.

yes

?- retract( raining).

yes

?- nice.

yes
```

Clauses of any form can be asserted or retracted. The next example illustrates that **retract** is also non-deterministic: a whole set of clauses can, through backtracking, be removed by a single **retract** goal. Let us assume that we have the following facts in the 'consulted' program:

```
fast( ann).
slow( tom).
slow( pat).
```

We can add a rule to this program, as follows:

```
?- assert(
        ( faster(X,Y) :-
            fast(X), slow(Y) ) ).

yes

?- faster( A, B).
```

```
A = ann
B = tom

?- retract( slow(X) ).

X = tom;
X = pat;
no

?- faster( ann, _).

no
```

Notice that when a rule is asserted, the syntax requires that the rule (as an argument to **assert**) be enclosed in parentheses.

When asserting a clause, we may want to specify the position at which the new clause is inserted to the database. The predicates **asserta** and **assertz** enable us to control the position of insertion. The goal

```
asserta( C)
```

adds C at the beginning of the database. The goal

```
assertz( C)
```

adds C at the end of the database. The following example illustrates these effects:

```
?- assert( p(a) ), assertz( p(b) ), asserta( p(c) ).

yes

?- p( X).

X = c;
X = a;
X = b
```

There is a relation between **consult** and **assertz**. Consulting a file can be defined in terms of **assertz** as follows: to consult a file, read each term (clause) in the file and assert it at the end of the database.

One useful application of **asserta** is to store already computed answers to questions. For example, let there be a predicate

```
solve( Problem, Solution)
```

defined in the program. We may now ask some question and request that the answer be remembered for future questions.

```
?- solve( problem1, Solution),
   asserta( solve( problem1, Solution) ).
```

If the first goal above succeeds then the answer (**Solution**) is stored and used, as any other clause, in answering further questions. The advantage of such a 'memoization' of answers is that a further question that matches the asserted fact will normally be answered much quicker than the first one. The result now will be simply retrieved as a fact, and not computed through a possibly time-consuming process.

An extension of this idea is to use asserting for generating all solutions in the form of a table of facts. For example, we can generate a table of products of all pairs of integers between 0 and 9 as follows: generate a pair of integers X and Y, compute Z is X*Y, assert the three numbers as one line of the product table, and then force the failure. The failure will cause, through backtracking, another pair of integers to be found and so another line tabulated, etc. The following procedure **maketable** implements this idea:

```
maketable :-
  L = [0,1,2,3,4,5,6,7,8,9],
  member( X, L),                % Choose first factor
  member( Y, L),                % Choose second factor
  Z is X*Y,
  assert( product(X,Y,Z) ),
  fail.
```

The question

```
?- maketable.
```

will, of course, not succeed, but it will, as a side effect, add the whole product table to the database. After that, we can ask for example, what pairs give the product 8:

```
?- product( A, B, 8).

A = 1
B = 8;

A = 2
B = 4;

...
```

A remark on the style of programming should be made at this stage. The foregoing examples illustrate some obviously useful applications of **assert** and **retract**. However, their use requires special care. Excessive and careless use of these facilities cannot be recommended as good programming style. By asserting and retracting we, in fact, modify the program. Therefore relations that

hold at some point will not be true at some other time. At different times the same questions receive different answers. A lot of asserting and retracting may thus obscure the meaning of the program and it may become hard to imagine what is true and what is not. The resulting behaviour of the program may become difficult to understand, difficult to explain and to trust.

Exercises

7.6 (a) Write a Prolog question to remove the whole **product** table from the database.

(b) Modify the question so that it only removes those entries where the product is 0.

7.7 Define the relation

copy(Term, Copy)

which will produce a copy of **Term** so that **Copy** is **Term** with all its variables renamed. This can be easily programmed by using **assert** and **retract**.

7.5 Control facilities

So far we have covered most of the extra control facilities except **repeat**. For completeness the complete set is presented here.

- *cut*, written as '!', prevents backtracking. It was introduced in Chapter 5.
- **fail** is a goal that always fails.
- **true** is a goal that always succeeds.
- **not(P)** is a type of negation that behaves exactly as if defined as:

  ```
  not( P) :- P, !, fail; true.
  ```

  Some problems with **cut** and **not** were discussed in detail in Chapter 5.
- **call(P)** invokes a goal P. It succeeds if P succeeds.
- **repeat** is a goal that always succeeds. Its special property is that it is non-deterministic; therefore, each time it is reached by backtracking it generates another alternative execution branch. **repeat** behaves as if defined by:

  ```
  repeat.
  repeat :- repeat.
  ```

  A typical way of using **repeat** is illustrated by the following procedure **dosquares** which reads a sequence of numbers and outputs their squares.

The sequence is concluded with the atom **stop** which serves as a signal for the procedure to terminate.

```
dosquares :-
  repeat,
  read( X),
  ( X = stop, !;
    Y is X*X, write(Y), fail).
```

7.6 *bagof, setof* and *findall*

We can generate, by backtracking, all the objects, one by one, that satisfy some goal. Each time a new solution is generated, the previous one disappears and is not accessible any more. However, sometimes we would prefer to have all the generated objects available together – for example, collected into a list. The built-in predicates **bagof** and **setof** serve this purpose; the predicate **findall** is sometimes provided instead.

The goal

```
bagof( X, P, L)
```

will produce the list L of all the objects X such that a goal P is satisfied. Of course, this usually makes sense only if X and P have some common variables. For example, let us assume that we have in the program a specification that classifies (some) letters into vowels and consonants:

```
class( a, vow).
class( b, con).
class( c, con).
class( d, con).
class( e, vow).
class( f, con).
```

Then we can obtain the list of all the consonants in this specification by the goal:

```
?- bagof( Letter, class( Letter, con), Letters).
Letters = [b,c,d,f]
```

If, in the above goal, we leave the class of a letter unspecified then we get, through backtracking, two lists of letters, each of them corresponding to each class:

```
?- bagof( Letter, class( Letter, Class), Letters).
```

```
Class = vow
Letters = [a,e];

Class = con
Letters = [b,c,d,f]
```

If there is no solution for P in the goal **bagof(X, P, L)** then the **bagof** goal simply fails. If the same object X is found repeatedly then all its occurrences will appear in L, which leads to duplicate items in L.

The predicate **setof** is similar to **bagof**. The goal

```
setof( X, P, L)
```

will again produce a list L of objects X that satisfy P. Only this time the list L will be ordered and duplicate items, if there are any, will be eliminated. The ordering of the objects is according to the alphabetical order or to the relation '<', if objects in the list are numbers. If the objects are structures then the principal functors are compared for the ordering. If these are equal then the left-most, top-most functors that are not equal in the terms compared decide.

There is no restriction on the kind of objects that are collected. So we can, for example, construct a list of pairs of the form

```
Class/Letter
```

so that the consonants come first ('con' is alphabetically before 'vow'):

```
?- setof( Class/Letter, class( Letter, Class), List).

List = [con/b, con/c, con/d, con/f, vow/a, vow/e]
```

Another predicate of this family, similar to **bagof**, is **findall**.

```
findall( X, P, L)
```

produces, again, a list of objects that satisfy P. The difference with respect to **bagof** is that *all* the objects X are collected regardless of (possibly) different solutions for variables in P that are not shared with X. This difference is shown in the following example:

```
?- findall( Letter, class( Letter, Class), Letters).

Letters = [a,b,c,d,e,f]
```

If there is no object X that satisfies P then **findall** will succeed with L = [].

If **findall** is not available as a built-in predicate in the implementation used then it can be easily programmed as follows. All solutions for P are generated by forced backtracking. Each solution is, when generated, immediately asserted into the database so that it is not lost when the next solution is found. After all the solutions have been generated and asserted, they have to

```
findall( X, Goal, Xlist) :-
  call( Goal),                    % Find a solution
  assertz( stack(X) ),            % Assert it
  fail;                           % Try to find more solutions
  assertz( stack(bottom) ),       % Mark end of solutions
  collect( Xlist).                % Collect the solutions

collect( L) :-
  retract( stack(X) ), !,         % Retract next solution
  ( X == bottom, !, L = [];       % End of solutions?
    L = [X | Rest], collect( Rest) ).   % Otherwise collect the rest
```

Figure 7.4 An implementation of the **findall** relation.

be collected into a list and retracted from the database. This whole process can be imagined as all the solutions generated forming a stack. Each newly generated solution is, by assertion, placed on top of this stack. When the solutions are collected the stack dissolves. Note, in addition, that the bottom of this stack has to be marked, for example, by the atom 'bottom' (which, of course, should be different from any solution that is possibly expected). An implementation of **findall** along these lines is shown as Figure 7.4.

Exercises

7.8 Use **bagof** to define the relation **powerset(Set, Subsets)** to compute the set of all subsets of a given set (all sets represented as lists).

7.9 Use **bagof** to define the relation

copy(Term, Copy)

such that **Copy** is **Term** with all its variables renamed.

Summary

- A Prolog implementation normally provides a set of built-in procedures to accomplish several useful operations that are not possible in pure Prolog. In this chapter, such a set of predicates, available in many Prolog implementations, was introduced.
- The type of a term can be tested by the following predicates:

var(X)	X is a (non-instantiated) variable
nonvar(X)	X is not a variable
atom(X)	X is an atom
integer(X)	X is an integer
atomic(X)	X is either an atom or an integer

- Terms can be constructed or decomposed:

  Term =.. [Functor | ArgumentList]
  functor(Term, Functor, Arity)
  arg(N, Term, Argument)
  name(Atom, CharacterCodes)

- A Prolog program can be viewed as a relational database that can be updated by the following procedures:

assert(Clause)	add **Clause** to the program
asserta(Clause)	add at the beginning
assertz(Clause)	add at the end
retract(Clause)	remove a clause that matches **Clause**

- All the objects that satisfy a given condition can be collected into a list by the predicates:

bagof(X, P, L)	L is the list of all X that satisfy condition P
setof(X, P, L)	L is the sorted list of all X that satisfy condition P
findall(X, P, L)	similar to **bagof**

- **repeat** is a control facility that generates an unlimited number of alternatives for backtracking

8 Programming Style and Technique

In this chapter we will review some general principles of good programming and discuss the following questions in particular: How to think about Prolog programs? What are elements of good programming style in Prolog? How to debug Prolog programs? How to make Prolog programs more efficient?

8.1 General principles of good programming

A fundamental question, related to good programming, is: What is a good program? Answering this question is not trivial as there are several criteria for judging how good a program is. Generally accepted criteria include the following:

- *Correctness* Above all, a good program should be correct. That is, it should do what it is supposed to do. This may seem a trivial, self-explanatory requirement. However, in the case of complex programs, correctness is often not attained. A common mistake when writing programs is to neglect this obvious criterion and pay more attention to other criteria, such as efficiency.

- *Efficiency* A good program should not needlessly waste computer time and memory space.

- *Transparency, readability* A good program should be easy to read and easy to understand. It should not be more complicated than necessary. Clever programming tricks that obscure the meaning of the program should be avoided. The general organization of the program and its layout help its readability.

- *Modifiability* A good program should be easy to modify and to extend. Transparency and modular organization of the program help modifiability.

- *Robustness* A good program should be robust. It should not crash immediately when the user enters some incorrect or unexpected data. The program should, in the case of such errors, stay 'alive' and behave reasonably (should report errors).

- *Documentation* A good program should be properly documented. The minimal documentation is the program's listing including sufficient program comments.

The importance of particular criteria depends on the problem and on the circumstances in which the program is written, and on the environment in which it is used. There is no doubt that correctness has the highest priority. The issues of transparency, modifiability, robustness and documentation are usually given, at least, as much priority as the issue of efficiency.

There are some general guidelines for practically achieving the above criteria. One important rule is to first *think* about the problem to be solved, and only then to start writing the actual code in the programming language used. Once we have developed a good understanding of the problem and the whole solution is well thought through, the actual coding will be fast and easy, and there is a good chance that we will soon get a correct program.

A common mistake is to start writing the code even before the full definition of the problem has been understood. A fundamental reason why early coding is bad practice is that the thinking about the problem and the ideas for a solution should be done in terms that are most relevant to the problem. These terms are usually far from the syntax of the programming language used, and they may include natural language statements and pictorial representation of ideas.

Such a formulation of the solution will have to be transformed into the programming language, but this transformation process may not be easy. A good approach is to use the principle of *stepwise refinement*. The initial formulation of the solution is referred to as the 'top-level solution', and the final program as the 'bottom-level solution'.

According to the principle of stepwise refinement, the final program is developed through a sequence of transformations, or 'refinements', of the solution. We start with the first, top-level solution and then proceed through a sequence of solutions; these are all equivalent, but each solution in the sequence is expressed in more detail. In each refinement step, concepts used in previous formulations are elaborated to greater detail and their representation gets closer to the programming language. It should be realized that refinement applies both to procedure definitions and to data structures. In the initial stages we normally work with more abstract, bulky units of information whose structure is refined later.

Such a strategy of top-down stepwise refinement has the following advantages:

- it allows for formulation of rough solutions in terms that are most relevant to the problem;
- in terms of such powerful concepts, the solution should be succint and simple, and therefore likely to be correct;
- each refinement step should be small enough so that it is intellectually manageable; if so, the transformation of a solution into a new, more

detailed representation is likely to be correct, and so is the resulting solution at the next level of detail.

In the case of Prolog we may talk about the stepwise refinement of *relations*. If the nature of the problem suggests thinking in algorithmic terms, then we can also talk about refinement of *algorithms*, adopting the procedural point of view on Prolog.

In order to properly refine a solution at some level of detail, and to introduce useful concepts at the next lower level, we need ideas. Therefore programming is creative, especially so for beginners. With experience, programming gradually becomes less of an art and more of a craft. But, nevertheless, a major question is: How do we get ideas? Most ideas come from experience, from similar problems whose solutions we know. If we do not know a direct programming solution, another similar problem could be helpful. Another source of ideas is everyday life. For example, if the problem is to write a program to sort a list of items we may get an idea from considering the question: How would I myself sort a set of exam papers according to the alphabetical order of students?

General principles of good programming outlined in this section are also known as the ingredients of 'structured programming', and they basically apply to Prolog as well. We will discuss some details with particular reference to Prolog in the following sections.

8.2 How to think about Prolog programs

One characteristic feature of Prolog is that it allows for both the procedural and declarative way of thinking about programs. The two approaches have been discussed in detail in Chapter 2, and illustrated by examples throughout the text. Which approach will be more efficient and practical depends on the problem. Declarative solutions are usually easier to develop, but may lead to an inefficient program.

During the process of developing a solution we have to find ideas for reducing problems to one or more easier subproblems. An important question is: How do we find proper subproblems? There are several general principles that often work in Prolog programming. These will be discussed in the following sections.

8.2.1 Use of recursion

The principle here is to split the problem into cases belonging to two groups:

(1) trivial, or 'boundary' cases;

(2) 'general' cases where the solution is constructed from solutions of (simpler) versions of the original problem itself.

In Prolog we use this technique all the time. Let us look at one more example: processing a list of items so that each item is transformed by the same transformation rule. Let this procedure be

maplist(List, F, NewList)

where **List** is an original list, F is a transformation rule (a binary relation) and **NewList** is the list of all transformed items. The problem of transforming **List** can be split into two cases:

(1) Boundary case: **List** = []

if **List** = [] then **NewList** = [], regardless of F

(2) General case: **List** = [X | Tail]

To transform a list of the form [X | Tail], do:
transform the list **Tail** obtaining **NewTail**, and
transform the item X by rule F obtaining **NewX**;
the whole transformed list is [NewX | NewTail].

In Prolog:

```
maplist( [], _, [] ).

maplist( [X | Tail], F, [NewX | NewTail] ) :-
  G =.. [F, X, NewX],
  call( G),
  maplist( Tail, F, NewTail).
```

One reason why recursion so naturally applies to defining relations in Prolog is that data objects themselves often have recursive structure. Lists and trees are such objects. A list is either empty (boundary case) or has a head and a tail that is itself a list (general case). A binary tree is either empty (boundary case) or it has a root and two subtrees that are themselves binary trees (general case). Therefore, to process a whole non-empty tree, we must do something with the root, and process the subtrees.

8.2.2 Generalization

It is often a good idea to generalize the original problem, so that the solution to the generalized problem can be formulated recursively. The original problem is then solved as a special case of its more general version. Generalization of a relation typically involves the introduction of one or more extra arguments. A major problem, which may require deeper insight into the problem, is how to find the right generalization.

As an example let us revisit the eight queens problem. The original

problem was to place eight queens on the chessboard so that they do not attack each other. Let us call the corresponding relation

eightqueens(Pos)

This is true if **Pos** is some representation of a position with eight non-attacking queens. A good idea in this case is to generalize the number of queens from eight to N. The number of queens now becomes the additional argument:

nqueens(Pos, N)

The advantage of this generalization is that there is an immediate recursive formulation of the **nqueens** relation:

(1) Boundary case: N = 0

To safely place zero queens is trivial.

(2) General case: N > 0

To safely place N queens on the board, satisfy the following:

- achieve a safe configuration of (N – 1) queens; and
- add the remaining queen so that she does not attack any other queen

Once the generalized problem has been solved, the original problem is easy:

```
eightqueens( Pos) :- nqueens( Pos, 8).
```

8.2.3 Using pictures

When searching for ideas about a problem, it is often useful to introduce some graphical representation of the problem. A picture may help us to perceive some essential relations in the problem. Then we just have to describe what we *see* in the picture in the programming language.

The use of pictorial representations is often useful in problem solving in general; it seems, however, that it works with Prolog particularly well. The following arguments explain why:

(1) Prolog is particularly suitable for problems that involve objects and relations about objects. Often, such problems can be naturally illustrated by graphs in which nodes correspond to objects and arcs correspond to relations.

(2) Structured data objects in Prolog are naturally pictured as trees.

(3) The declarative meaning of Prolog facilitates the translation of pictorial representations into Prolog because, in principle, the order in which the

picture is described does not matter. We just put what we see into the program in any order. (For practical reasons of the program's efficiency this order will possibly have to be polished later.)

8.3 Programming style

The purpose of conforming to some stylistic conventions is:

- to reduce the danger of programming errors; and
- to produce programs that are readable and easy to understand, easy to debug and to modify.

We will review here some ingredients of good programming style in Prolog: some general rules of good style, tabular organization of long procedures and commenting.

8.3.1 Some rules of good style

- Program clauses should be short. Their body should typically contain no more than a few goals.
- Procedures should be short because long procedures are hard to understand. However, long procedures are acceptable if they have some uniform structure (this will be discussed later in this section).
- Mnemonic names for procedures and variables should be used. Names should indicate the meaning of relations and the role of data objects.
- The layout of programs is important. Spacing, blank lines and indentation should be consistently used for the sake of readability. Clauses about the same procedure should be clustered together; there should be blank lines between clauses (unless, perhaps, there are numerous facts about the same relation); each goal can be placed on a separate line. Prolog programs sometimes resemble poems for the aesthetic appeal of ideas and form.
- Stylistic conventions of this kind may vary from program to program as they depend on the problem and personal taste. It is important, however, that the same conventions are used consistently throughout the whole program.
- The cut operator should be used with care. Cut should not be used if it can be easily avoided. It is better to use, where possible, 'green cuts' rather than 'red cuts'. As discussed in Chapter 5, a cut is called 'green' if it can be removed without altering the declarative meaning of the clause. The use of 'red cuts' should be restricted to clearly defined constructs such as **not** or the selection between alternatives. An example of the latter construct is:

  if **Condition** *then* **Goal1** *else* **Goal2**

This translates into Prolog, using cut, as:

```
Condition, !,          % Condition true?
Goal1;                 % If yes then Goal1
Goal2                  % Otherwise Goal2
```

- The **not** operator can also lead to surprising behaviour, as it is related to cut. We have to be well aware of how **not** is defined in Prolog. However, if there is a dilemma between **not** and cut, the former is perhaps better than some obscure construct with cut.
- Program modification by **assert** and **retract** can grossly degrade the transparency of the program's behaviour. In particular, the same program will answer the same question differently at different times. In such cases, if we want to reproduce the same behaviour we have to make sure that the whole previous state, which was modified by assertions and retractions, is completely restored.
- The use of a semicolon may obscure the meaning of a clause. The readability can sometimes be improved by splitting the clause containing the semicolon into more clauses; but this will, possibly, be at the expense of the length of the program and its efficiency.

To illustrate some points of this section consider the relation

merge(List1, List2, List3)

where **List1** and **List2** are ordered lists that merge into **List3**. For example:

merge([2,4,7], [1,3,4,8], [1,2,3,4,4,7,8])

The following is an implementation of **merge** in bad style:

```
merge( List1, List2, List3) :-
  List1 = [], !, List3 = List2;        % First list empty
  List2 = [], !, List3 = List1;        % Second list empty
  List1 = [X | Rest1],
  List2 = [Y | Rest2],
  ( X < Y, !,
    Z = X,                             % Z is head of List3
    merge( Rest1, List2, Rest3);
    Z = Y,
    merge( List1, Rest2, Rest3) ),
  List3 = [Z | Rest3].
```

Here is a better version which avoids semicolons:

```
merge( [], List, List).

merge( List, [], List).
```

```
merge( [X | Rest1], [Y | Rest2], [X | Rest3] ) :-
  X < Y, !,
  merge( Rest1, [Y | Rest2], Rest3).

merge( List1, [Y | Rest2], [Y | Rest3] ) :-
  merge( List1, Rest2, Rest3).
```

8.3.2 Tabular organization of long procedures

Long procedures are acceptable if they have some uniform structure. Typically, such a form is a set of facts when a relation is effectively defined in the tabular form. Advantages of such an organization of a long procedure are:

- Its structure is easily understood.
- Incrementability: it can be refined by simply adding new facts.
- It is easy to check and correct or modify (by simply replacing some fact independently of other facts).

8.3.3 Commenting

Program comments should explain in the first place what the program is about and how to use it, and only then the details of the solution method used and other programming details. The main purpose of comments is to enable the user to use the program, to understand it and to possibly modify it. Comments should describe, in the shortest form possible, everything that is essential to these ends. Undercommenting is a usual fault, but a program can also be overcommented. Explanation of details that are obvious from the program code itself is only a needless burden to the program.

Long passages of comments should precede the code they refer to, while short comments should be interspersed with the code itself. Information that should, in general, be included in comments comprises the following:

- What the program does, how it is used (for example, what goal is to be invoked and what are the expected results), examples of using the program.
- What are top-level predicates?
- How are main concepts (objects) represented?
- Execution time and memory requirements of the program.
- What are the program's limitations?
- Are there any special system-dependent features used?
- What is the meaning of the predicates in the program? What are their arguments? Which arguments are 'input' and which are 'output', if known? (Input arguments have fully specified values, without uninstantiated variables, when the predicate is called.)
- Algorithmic and implementation details.

8.4 Debugging

When a program does not do what it is expected to do the main problem is to locate the error(s). It is easier to locate an error in a part of the program (or a module) than in the program as a whole. Therefore, a good principle of debugging is to start by testing smaller units of the program, and when these can be trusted, to start testing bigger modules or the whole program.

Debugging in Prolog is facilitated by two things: first, Prolog is an interactive language so any part of the program can be directly invoked by a proper question to the Prolog system; second, Prolog implementations usually provide special debugging aids. As a result of these two features, debugging of Prolog programs can, in general, be done far more efficiently than in most other programming languages.

The basis for debugging aids is *tracing*. 'Tracing a goal' means that the information regarding the goal's satisfaction is displayed during execution. This information includes:

- Entry information: the predicate name and the values of arguments when the goal is invoked.
- Exit information: in the case of success, the values of arguments that satisfy the goal; otherwise an indication of failure.
- Re-entry information: invocation of the same goal caused by backtracking.

Between entry and exit, the trace information for all the subgoals of this goal can be obtained. So we can trace the execution of our question all the way down to the lowest level goals until facts are encountered. Such detailed tracing may turn out to be impractical due to the excessive amount of tracing information; therefore, the user can specify selective tracing. There are two selection mechanisms: first, suppress tracing information beyond a certain level; second, trace only some specified subset of predicates, and not all of them.

Such debugging aids are activated by system-dependent built-in predicates. A typical subset of such predicates is as follows:

trace

triggers exhaustive tracing of goals that follow.

notrace

stops further tracing.

spy(P)

specifies that a predicate P be traced. This is used when we are particularly interested in the named predicate and want to avoid tracing information from

other goals (either above or below the level of a call of P). Several predicates can be simultaneously active for 'spying'.

nospy(P)

stops 'spying' P.

Tracing beyond a certain depth can be suppressed by special commands during execution. There may be several other debugging commands available, such as returning to a previous point of execution. After such a return we can, for example, repeat the execution at a greater detail of tracing.

8.5 Efficiency

There are several aspects of efficiency, including the most common ones, execution time and space requirements of a program. Another aspect is the time needed by the programmer to develop the program.

The traditional computer architecture is not particularly suitable for the Prolog style of program execution – that is, satisfying a list of goals. Therefore, the limitations of time and space may be experienced earlier in Prolog than in many other programming languages. Whether this will cause difficulties in a practical application depends on the problem. The issue of time efficiency is practically meaningless if a Prolog program that is run a few times per day takes 1 second of CPU time and a corresponding program in some other language, say Fortran, takes 0.1 seconds. The difference in efficiency will perhaps matter if the two programs take 50 minutes and 5 minutes respectively.

On the other hand, in many areas of application Prolog will greatly reduce the program development time. Prolog programs will, in general, be easier to write, to understand and to debug than in traditional languages. Problems that gravitate toward the 'Prolog domain' involve symbolic, non-numeric processing, structured data objects and relations between them. In particular, Prolog has been successfully applied in areas, such as symbolic solving of equations, planning, databases, general problem solving, prototyping, implementation of programming languages, discrete and qualitative simulation, architectural design, machine learning, natural language understanding, expert systems, and other areas of artificial intelligence. On the other hand, numerical mathematics is an area for which Prolog is not a natural candidate.

With respect to the execution efficiency, executing a *compiled* program is generally more efficient than *interpreting* the program. Therefore, if the Prolog system contains both an interpreter and a compiler, then the compiler should be used if efficiency is critical.

If a program suffers from inefficiency then it can often be radically improved by improving the algorithm itself. However, to do this, the procedural aspects of the program have to be studied. A simple way of improving the executional efficiency is to find a better ordering of clauses of procedures,

and of goals in the bodies of procedures. Another relatively simple method is to provide guidance to the Prolog system by means of cuts.

Ideas for improving the efficiency of a program usually come from a deeper understanding of the problem. A more efficient algorithm can, in general, result from improvements of two kinds:

- Improving search efficiency by avoiding unnecessary backtracking and stopping the execution of useless alternatives as soon as possible.
- Using more suitable data structures to represent objects in the program, so that operations on objects can be implemented more efficiently.

We will study both kinds of improvements by looking at examples. Yet another technique of improving efficiency will be illustrated by an example. This technique is based on asserting into the database intermediate results that are likely to be needed again in the future computation. Instead of repeating the computation, such results are simply retrieved as already known facts.

8.5.1 Improving the efficiency of an eight queens program

As a simple example of improving the search efficiency let us revisit the eight queens problem (see Figure 4.7). In this program, the Y-coordinates of the queens are found by successively trying, for each queen, the integers between 1 and 8. This was programmed as the goal:

```
member( Y, [1,2,3,4,5,6,7,8] )
```

The way that **member** works is that Y = 1 is tried first, and then Y = 2, Y = 3, etc. As the queens are placed one after another in adjacent columns on the board, it is obvious that this order of trials is not the most appropriate. The reason for this is that the queens in adjacent columns will attack each other if they are not placed at least two squares apart in the vertical direction. According to this observation, a simple attempt to improve the efficiency is to rearrange the candidate coordinate values. For example:

```
member( Y, [1,5,2,6,3,7,4,8] )
```

This minor change will reduce the time needed to find the first solution by a factor of 3 or 4.

In the next example, a similarly simple idea of reordering will convert a practically unacceptable time complexity into a trivial one.

8.5.2 Improving the efficiency in a map colouring program

The map colouring problem is to assign each country in a given map one of four given colours in such a way that no two neighbouring countries are painted with the same colour. There is a theorem which guarantees that this is always possible.

Let us assume that a map is specified by the neighbour relation

ngb(Country, Neighbours)

where **Neighbours** is the list of countries bordering on **Country**. So the map of Europe, with 20 countries, would be specified (in alphabetical order) as:

```
ngb( albania, [greece, yugoslavia] ).
ngb( andorra, [france, spain] ).
ngb( austria, [czechoslovakia, hungary, italy, liechtenstein,
               switzerland, westgermany, yugoslavia] ).
...
```

Let a solution be represented as a list of pairs of the form

Country/Colour

which specifies a colour for each country in a given map. For the given map, the names of countries are fixed in advance, and the problem is to find the values for the colours. Thus, for Europe, the problem is to find a proper instantiation of variables C1, C2, C3 , etc. in the list:

[albania/C1, andorra/C2, austria/C3, ...]

Now let us define the predicate

colours(Country_colour_list)

which is true if the **Country_colour_list** satisfies the map colouring constraint with respect to a given **ngb** relation. Let the four colours be yellow, blue, red and green. The condition that no two neighbouring countries are of the same colour can be formulated in Prolog as follows:

```
colours( [] ).

colours( [Country/Colour | Rest] ) :-
  colours( Rest),
  member( Colour, [yellow, blue, red, green] ),
  not( member( Country1/Colour, Rest), neighbour( Country, Country1) ).

neighbour( Country, Country1) :-
  ngb( Country, Neighbours),
  member( Country1, Neighbours).
```

Here, **member(X,L)** is, as usual, the list membership relation. This will work well for simple maps, with a small number of countries. Europe might be problematic, however. Assuming that the built-in predicate **setof** is available,

one attempt to colour Europe could be as follows. First, let us define an auxiliary relation

```
country( C) :- ngb( C, _).
```

Then the question for colouring Europe can be formulated as:

```
?- setof( Cntry/Colour, country( Cntry), CountryColourList),
   colours( CountryColourList).
```

The **setof** goal will construct a template country/colour list for Europe in which uninstantiated variables stand for colours. Then the **colours** goal is supposed to instantiate the colours. However, this attempt will probably fail because of inefficiency.

A detailed study of the way Prolog tries to satisfy the **colours** goal reveals the source of inefficiency. Countries in the country/colour list are arranged in alphabetical order, and this has nothing to do with their geographical arrangement. The order in which the countries are assigned colours corresponds to the order in the list (starting at the end), which is in our case independent of the **ngb** relation. So the colouring process starts at some end of the map, continues at some other end, etc., moving around more or less randomly. This may easily lead to a situation in which a country that is to be coloured is surrounded by many other countries, already painted with all four available colours. Then backtracking is necessary which leads to inefficiency.

It is clear, then, that the efficiency depends on the order in which the countries are coloured. Intuition suggests a simple colouring strategy that should be better than random: start with some country that has many neighbours, and then proceed to the neighbours, then to the neighbours of neighbours, etc. For Europe, then, West Germany (having most neighbours, 9) is a good candidate to start with. Of course, when the template country/colour list is constructed, West Germany has to be put at the end of the list and other countries have to be added at the front of the list. In this way the colouring algorithm, which starts at the rear end, will commence with West Germany and proceed from there from neighbour to neighbour.

Such a country/colour template dramatically improves the efficiency with respect to the original, alphabetical order, and possible colourings for the map of Europe will be now produced without difficulty.

We can construct a properly ordered list of countries manually, but we do not have to. The following procedure, **makelist**, does it. It starts the construction with some specified country (West Germany in our case) and collects the countries into a list called **Closed**. Each country is first put into another list, called **Open**, before it is transferred to **Closed**. Each time that a country is transferred from **Open** to **Closed**, its neighbours are added to **Open**.

```
makelist( List) :-
  collect( [westgermany], [], List).

collect( [], Closed, Closed).          % No more candidates for Closed
```

```
collect( [X | Open], Closed, List) :-
  member( X, Closed), !,                % X has already been collected?
  collect( Open, Closed, List).         % Discard X

collect( [X | Open], Closed, List) :-
  ngb( X, Ngbs),                        % Find X's neighbours
  conc( Ngbs, Open, Open1),             % Put them to Open1
  collect( Open1, [X | Closed], List).  % Collect the Rest
```

The conc relation is, as usual, the list concatenation relation.

8.5.3 Improving the efficiency of a list concatenation by a better data structure

In our programs so far, the concatenation of lists has been programmed as:

```
conc( [], L, L).

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

This is inefficient when the first list is long. The following example explains why:

```
?- conc( [a,b,c], [d,e], L).
```

This produces the following sequence of goals:

```
conc( [a,b,c], [d,e], L)
  conc( [b,c], [d,e], L')          where L = [a | L']
    conc( [c], [d,e], L'')           where L' = [b | L'']
      conc( [], [d,e], L''')           where L'' = [c | L''']
        true                             where L''' = [d,e]
```

From this it is clear that the program in effect scans all of the first list, until the empty list is encountered.

But could we not simply skip the whole of the first list in a single step and append the second list, instead of gradually working down the first list? To do this, we need to know where the end of a list is; that is, we need another representation of lists. One solution is to represent a list by a pair of lists. For example, the list

```
[a,b,c]
```

can be represented by the two lists:

```
L1 = [a,b,c,d,e]
L2 = [d,e]
```

Such a pair of lists, which we will for brevity choose to write as **L1-L2**, represents the 'difference' between L1 and L2. This of course only works under the condition that L2 is a suffix of L1. Note that the same list can be represented by several 'difference pairs'. So the list **[a,b,c]** can be represented by

```
    [a,b,c]-[]
or
    [a,b,c,d,e]-[d,e]
or
    [a,b,c,d,e | T]-[d,e | T]
or
    [a,b,c | T]-T
```

where T is any list, etc. The empty list is represented by any pair **L-L**.

As the second member of the pair indicates the end of the list, the end is directly accessible. This can be used for an efficient implementation of concatenation. The method is illustrated in Figure 8.1. The corresponding concatenation relation translates into Prolog as the fact:

```
concat( A1-Z1, Z1-Z2, A1-Z2).
```

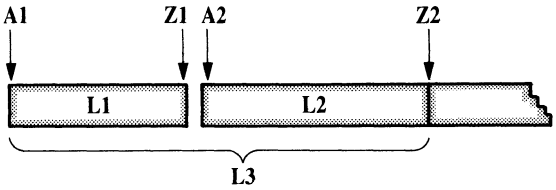


Figure 8.1 Concatenation of lists represented by difference pairs. **L1** is represented by **A1-Z1**, **L2** by **A2-Z2**, and the result **L3** by **A1-Z2** when **Z1 = A2** must be true.

Let us use **concat** to concatenate the lists **[a,b,c]**, represented by the pair **[a,b,c | T1]-T1**, and the list **[d,e]**, represented by **[d,e | T2]-T2**:

```
?- concat( [a,b,c | T1]-T1, [d,e | T2]-T2, L).
```

The concatenation is done just by matching this goal with the clause about **concat**, giving:

```
T1 = [d,e | T2]
L  = [a,b,c,d,e | T2]-T2
```

8.5.4 Improving the efficiency by asserting derived facts

Sometimes during computation the same goal has to be satisfied again and

again. As Prolog has no special mechanism to discover such situations whole computation sequences are repeated.

As an example consider a program to compute the Nth Fibonacci number for a given N. The Fibonacci sequence is

1, 1, 2, 3, 5, 8, 13, ...

Each number in the sequence, except for the first two, is the sum of the previous two numbers. We will define a predicate

fib(N, F)

to compute, for a given N, the Nth Fibonacci number, F. We count the numbers in the sequence starting with N = 1. The following **fib** program deals first with the first two Fibonacci numbers as two special cases, and then specifies the general rule about the Fibonacci sequence:

```
fib( 1, 1).                    % 1st Fibonacci number

fib( 2, 1).                    % 2nd Fibonacci number

fib( N, F) :-                  % Nth Fib. number, N > 2
  N > 2,
  N1 is N-1, fib( N1, F1),
  N2 is N-2, fib( N2, F2),
  F is F1 + F2.                % Nth number is the sum of
                               % its two predecessors
```

This program tends to redo parts of the computation. This is easily seen if we trace the execution of the following goal:

```
?- fib( 6, F).
```

Figure 8.2 illustrates the essence of this computational process. For example, the third Fibonacci number, *f(3)*, is needed in three places and the same computation is repeated each time.

This can be easily avoided by remembering each newly computed Fibonacci number. The idea is to use the built-in procedure **assert** and to add these (intermediate) results as facts to the database. These facts have to precede other clauses about **fib** to prevent the use of the general rule in cases where the result is already known. The modified procedure, **fib2**, differs from **fib** only in this assertion:

```
fib2( 1, 1).                   % 1st Fibonacci number

fib2( 2, 1).                   % 2nd Fibonacci number

fib2( N, F) :-                 % Nth Fib. number, N > 2
  N > 2,
```

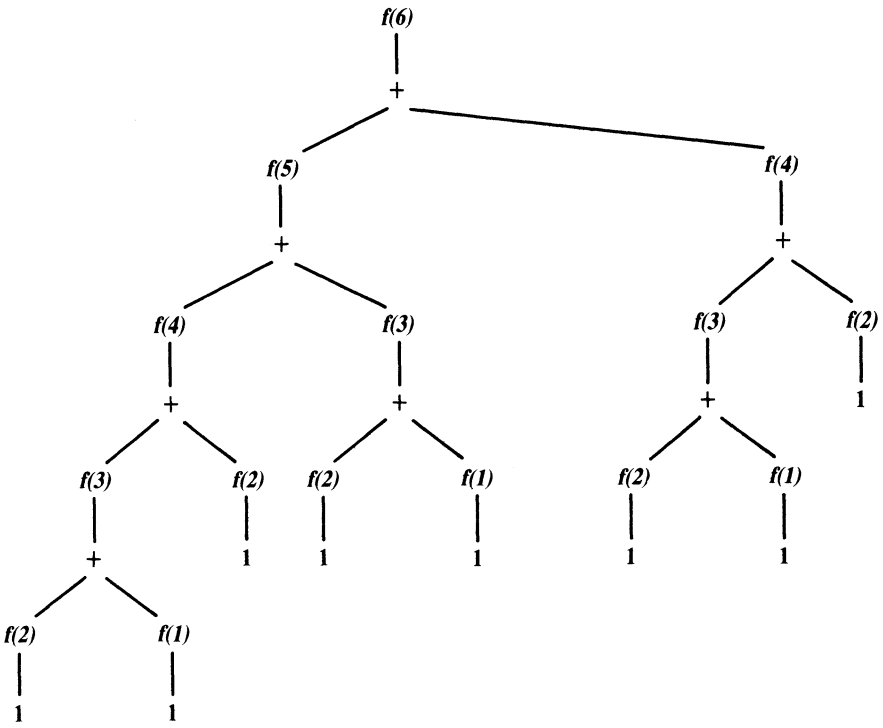


Figure 8.2 Computation of the 6th Fibonacci number by procedure **fib**.

```
N1 is N-1, fib2( N1, F1),
N2 is N-2, fib2( N2, F2),
F is F1 + F2,
asserta( fib2( N, F) ).          % Remember Nth number
```

This program will try to answer any **fib2** goal by first looking at stored facts about this relation, and only then resort to the general rule. As a result, when a goal **fib2(N, F)** is executed all Fibonacci numbers, up to the Nth number, will get tabulated. Figure 8.3 illustrates the computation of the 6th Fibonacci number by **fib2**. A comparison with Figure 8.2 shows the saving in the computational complexity. For greater N, the savings would be much more substantial.

Asserting intermediate results is a standard technique for avoiding repeated computations. It should be noted, however, that in the case of Fibonacci numbers we can also avoid repeated computation by using another algorithm, rather than by asserting intermediate results. This other algorithm will lead to a program that is more difficult to understand, but more efficient to execute. The idea this time is not to define the Nth Fibonacci number simply as the sum of its two predecessors and leave the recursive calls to unfold the whole computation

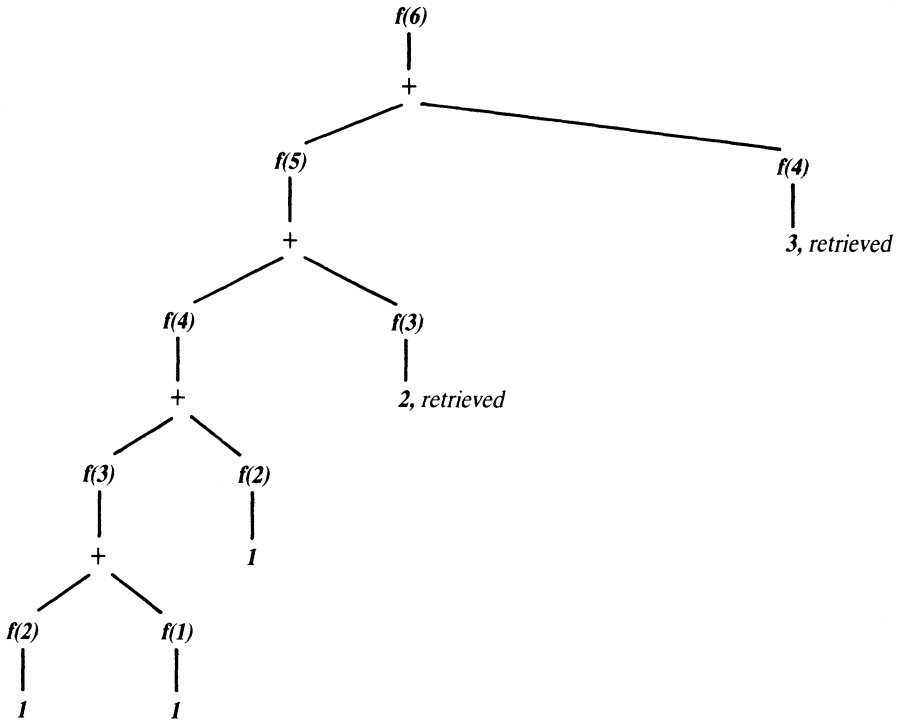


Figure 8.3 Computation of the 6th Fibonacci number by procedure **fib2**, which remembers previous results. This saves some computation in comparison with **fib**, see Figure 8.2.

'downwards' to the two initial Fibonacci numbers. Instead, we can work 'upwards', starting with the initial two numbers, and compute the numbers in the sequence one by one in the forward direction. We have to stop when we have computed the Nth number. Most of the work in such a program is done by the procedure:

forwardfib(M, N, F1, F2, F)

Here, F1 and F2 are the (M - 1)st and Mth Fibonacci numbers, and F is the Nth Fibonacci number. Figure 8.4 helps to understand the **forwardfib** relation. According to this figure, **forwardfib** finds a sequence of transformations to reach a final configuration (when M = N) from a given starting configuration. When **forwardfib** is invoked, all the arguments except F have to be instantiated, and M has to be less or equal to N. The program is:

```
fib3( N, F) :-
  forwardfib( 2, N, 1, 1, F).          % The first two Fib. numbers are 1

forwardfib( M, N, F1, F2, F2) :-
  M >= N.                              % Nth Fibonacci number reached
```

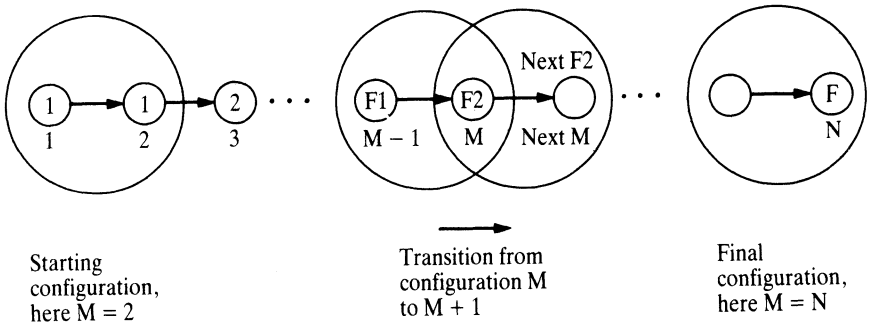


Figure 8.4 Relations in the Fibonacci sequence. A 'configuration', depicted by a large circle, is defined by three things: an index M and two consecutive Fibonacci numbers $f(M-1)$ and $f(M)$.

```
forwardfib( M, N, F1, F2, F) :-
  M < N,                          % Nth number not yet reached
  NextM is M + 1,
  NextF2 is F1 + F2,
  forwardfib( NextM, N, F2, NextF2, F).
```

Exercises

8.1 Procedures **sub1**, **sub2** and **sub3**, shown below, all implement the sublist relation. **sub1** is a more procedural definition whereas **sub2** and **sub3** are written in a more declarative style. Study the behaviour, with reference to efficiency, of these three procedures on some sample lists. Two of them behave nearly equivalently and have similar efficiency. Which two? Why is the remaining one less efficient?

```
sub1( List, Sublist) :-
  prefix( List, Sublist).
sub1( [_ | Tail], Sublist) :-
  sub1( Tail, Sublist).           % Sublist is sublist of Tail
prefix( _, [] ).
prefix( [X | List1], [X | List2] ) :-
  prefix( List1, List2).
sub2( List, Sublist) :-
  conc( List1, List2, List),
  conc( List3, Sublist, List1).
sub3( List, Sublist) :-
  conc( List1, List2, List),
  conc( Sublist, _, List2).
```

8.2 Define the relation

add_at_end(List, Item, NewList)

to add **Item** at the end of **List** producing **NewList**. Let both lists be represented by difference pairs.

8.2 Define the relation

reverse(List, ReversedList)

where both lists are represented by difference pairs.

8.4 Rewrite the **collect** procedure of Section 8.5.2 using difference pair representation for lists so that the concatenation can be done more efficiently.

Summary

- There are several criteria for evaluating programs:

  correctness
  efficiency
  transparency, readability
  modifiability
  robustness
  documentation

- The principle of *stepwise refinement* is a good way of organizing the program development process. Stepwise refinement applies to relations, algorithms and data structures.
- In Prolog, the following techniques often help to find ideas for refinements:

  Using recursion: identify boundary and general cases of a recursive definition.

  Generalization: consider a more general problem that may be easier to solve than the original one.

  Using pictures: graphical representation may help to identify important relations.

- It is useful to conform to some stylistic conventions to reduce the danger of programming errors, make programs easier to read, debug and modify.
- Prolog systems usually provide program debugging aids. Trace facilities are most useful.

- There are many ways of improving the efficiency of a program. Simple techniques include:

  reordering of goals and clauses

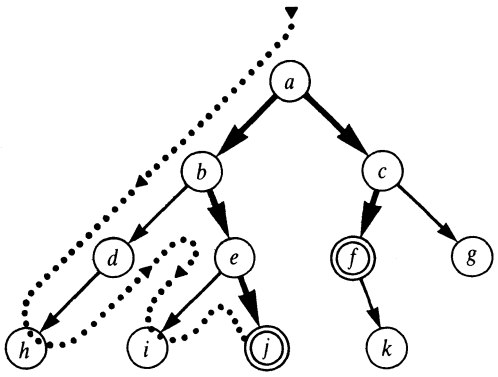
  controlling backtracking by inserting cuts

  remembering (by **assert**) solutions that would otherwise be computed again

  More sophisticated and radical techniques aim at better algorithms (improving search efficiency in particular) and better data structures.

PART TWO

PROLOG IN ARTIFICIAL INTELLIGENCE



9 Operations on Data Structures

One fundamental question in programming is how to represent complex data objects, such as sets, and efficiently implement operations on such objects. In general, we talk about selecting a proper data structure. The theme of this chapter is some frequently used data structures that belong to three big families: lists, trees and graphs. We will examine ways of representing these structures in Prolog, and develop programs for some operations on these structures, such as sorting a list, representing data sets by tree structures, storing data in trees and retrieving data from trees, path finding in graphs, etc. We will study several examples because these operations are extremely instructive for programming in Prolog.

9.1 Representing and sorting lists

9.1.1 Remarks on alternative representations of lists

The special Prolog notation for lists was introduced in Chapter 3 and is used throughout the text. This notation is, of course, only one way for representing lists. A *list* is, in general, defined as a structure that is either

- *empty*, or
- it consists of a *head* and a *tail;* the tail itself has to be a list too.

Therefore, to represent a list in general, we only need two things: a special symbol to represent the empty list, and a functor to combine a head and a tail. Thus one way to represent lists could be to choose

donothing

as the symbol that denotes the empty list, and the atom

then

as an infix operator to construct a list from a given head and tail. We can

declare this operator, for example, as:

```
:- op( 500, xfy, then).
```

The list

```
[ enter, sit, eat]
```

can then be written in this notation as:

```
enter then sit then eat then donothing
```

It is important to notice that the special Prolog notation for lists and the alternative representations amount essentially to the same representation if properly abstracted. Typical operations on lists, such as

```
member( X, L)
conc( L1, L2, L3)
del( X, L1, L2)
```

that have been programmed for the special list notation, can be straightforwardly rewritten into other, user-chosen representations. For example, the conc relation translates into the 'then–donothing' notation as follows. The definition that we have been using is:

```
conc( [], L, L).

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

In our 'then–donothing' notation this becomes:

```
conc( donothing, L, L).

conc( X then L1, L2, X then L3) :-
  conc( L1, L2, L3).
```

This example illustrates how our definitions of relations regarding lists are easily generalized to the whole class of structures of this type. What notation for lists will be actually used in the program should be decided according to the meaning of the lists. If, for example, a list simply means a set of items, then the Prolog list notation is most convenient as it immediately indicates what the programmer had in mind. On the other hand, certain kinds of expressions can also be viewed as lists. For example, to represent a conjunc-

tion expression in propositional logic, a suitable list-like representation would be:

- **true** corresponds to the empty list
- & is the operator to combine the head and the tail, defined for example as:

  :- op(300, xfy, &)

Then the conjunction of a, b and c would be represented by:

a & b & c & true

All these examples of different notations for lists are essentially based on the same structure for lists. On the other hand, in Chapter 8 we discussed another, essentially different representation of lists, which affects computation on lists. The trick there was to represent the list by a pair of lists. The list represented by such a pair of lists is the difference between the two lists. It was shown that the difference-pair representation facilitates very efficient implementation of list concatenation.

The discussion in this section also further illuminates the difference between operators in mathematics and the operator notation in Prolog. In mathematics, actions are associated with operators whereas in Prolog the operator notation is simply used for representing structures.

Exercises

9.1 Define the predicate

list(Object)

to recognize whether **Object** is a standard Prolog list.

9.2 Define the list membership relation for lists represented in the 'then–donothing' notation of this section.

9.3 Define the relation

convert(StandardList, List)

for converting between lists, written in the standard list notation and the 'then–donothing' notation. For example:

convert([a,b], a then b then donothing)

9.4 Generalize the above **convert** relation to general representation of lists. The specific representation desired is then specified by a symbol to

denote the empty list, and a functor to combine the head and the tail. To this end **convert** needs two additional arguments:

convert(StandardList, NewList, Functor, EmptyList)

Examples of using this relation are:

```
?- convert( [a,b], L, then, donothing)
L = a then b then donothing

?- convert( [a,b,c], L, +, 0).
L = a+(b+(c+0) )
```

9.1.2 Sorting lists

Sorting is a frequent operation. A list can be sorted if there is an ordering relation between the items in the list. We will for the purpose of this discussion assume that there is an ordering relation

gt(X, Y)

meaning that X is *greater than* Y, whatever 'greater than' means. If our items are numbers then the **gt** relation will perhaps be defined as:

```
gt( X, Y) :- X > Y.
```

If the items are atoms then the **gt** relation can correspond to the alphabetical order.

Let

sort(List, Sorted)

denote a relation where **List** is a list of items and **Sorted** is a list of the same items sorted in the ascending order according to the **gt** relation. We will develop three definitions of this relation in Prolog, based on different ideas for sorting a list. The first idea is as follows:

To sort a list, **List**:

- Find two adjacent elements, X and Y, in **List** such that **gt(X, Y)** and swap X and Y in **List**, obtaining **List1**; then sort **List1**.
- If there is no pair of adjacent elements, X and Y, in **List** such that **gt(X, Y)**, then **List** is already sorted.

The purpose of swapping two elements, X and Y, that occur out of order, is

that after the swapping the new list is closer to a sorted list. After a sufficient amount of swapping we should end up with all the elements in order. This principle of sorting is known as *bubble sort*. The corresponding Prolog procedure will be therefore called **bubblesort**:

```
bubblesort( List, Sorted) :-
  swap( List, List1), !,                % A useful swap in List?
  sort( List1, Sorted).

bubblesort( Sorted, Sorted).            % Otherwise list is already sorted

swap( [X, Y | Rest], [Y, X | Rest] ) :- % Swap first two elements
  gt( X, Y).

swap( [Z | Rest], [Z | Rest1] ) :-      % Swap elements in tail
  swap( Rest, Rest1).
```

Another simple sorting algorithm is *insertion sort*, which is based on the following idea.

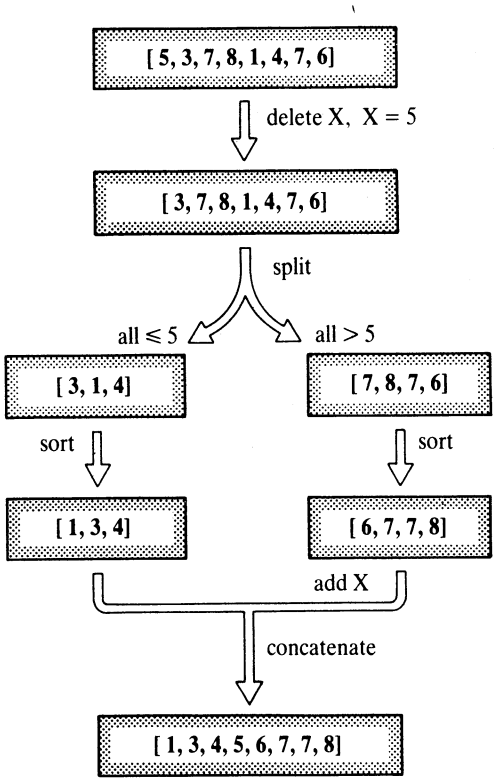


Figure 9.1 Sorting a list by *quicksort*.

To sort a non-empty list, L = [X | T]:

(1) Sort the tail T of L.

(2) Insert the head, X, of L into the sorted tail at such a position that the resulting list is sorted. The result is the whole sorted list.

This translates into Prolog as the following **insertsort** procedure:

```
insertsort( [], [] ).

insertsort( [X | Tail], Sorted) :-
  insertsort( Tail, SortedTail),          % Sort the tail
  insert( X, SortedTail, Sorted).         % Insert X at proper place

insert( X, [Y | Sorted], [Y | Sorted1] ) :- -
  gt( X, Y), !,
  insert( X, Sorted, Sorted1).

insert( X, Sorted, [X | Sorted] ).
```

The sorting procedures **bubblesort** and **insertsort** are simple, but inefficient. Of the two procedures, insertion sort is the more efficient one. However, the average time that **insertsort** requires for sorting a list of length n grows proportionally to n^2. For long lists, therefore, a much better sorting algorithm is *quicksort*. This is based on the following idea, which is illustrated in Figure 9.1:

To sort a non-empty list, L:

(1) Delete some element X from L and split the rest of L into two lists, called **Small** and **Big**, as follows: all elements in L that are greater than X belong to **Big**, and all others to **Small**.

(2) Sort **Small** obtaining **SortedSmall**.

(3) Sort **Big** obtaining **SortedBig**.

(4) The whole sorted list is the concatenation of **SortedSmall** and **[X | SortedBig]**.

If the list to be sorted is empty then the result of sorting is also the empty list. A Prolog implementation of quicksort is shown in Figure 9.2. A particular detail of this implementation is that the element, X, that is deleted from L is always simply the head of L. The splitting is programmed as a four-argument relation:

split(X, L, Small, Big)

The time complexity of this algorithm depends on how lucky we are when

```
quicksort( [], [] ).

quicksort( [X | Tail], Sorted) :-
  split( X, Tail, Small, Big),
  quicksort( Small, SortedSmall),
  quicksort( Big, SortedBig),
  conc( SortedSmall, [X | SortedBig], Sorted).

split( X, [], [], [] ).

split( X, [Y | Tail], [Y | Small], Big) :-
  gt( X, Y), !,
  split( X, Tail, Small, Big).

split( X, [Y | Tail], Small, [Y | Big] ) :-
  split( X, Tail, Small, Big).

conc( [], L, L).

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

Figure 9.2 Quicksort.

splitting the list to be sorted. If the list is split into two lists of approximately equal lengths then the time complexity of this sorting procedure is of the order $n \log n$ where n is the length of the list to be sorted. If, on the contrary, splitting always results in one list far bigger than the other, then the complexity is in the order of n^2. Analysis would show that the average performance of quicksort is, fortunately, closer to the best case than to the worst case.

The program in Figure 9.2 can be further improved by a better implementation of the concatenation operation. Using the difference-pair representation of lists, introduced in Chapter 8, concatenation is reduced to triviality. To use this idea in our sorting procedure, the lists in the program of Figure 9.2 can be represented by pairs of lists of the form A-Z as follows:

S1 is represented by A1-Z1
S2 is represented by A2-Z2

Then the concatenation of the lists S1 and [X | S2] corresponds to the concatenation of pairs

A1-Z1 and [X | A2]-Z2

The resulting concatenated list is represented by

A1-Z2 where Z1 = [X | A2]

```
quicksort( List, Sorted) :-
  quicksort2( List, Sorted-[] ).

quicksort2( [], Z-Z).

quicksort2( [X | Tail], A1-Z2) :-
  split( X, Tail, Small, Big),
  quicksort2( Small, A1-[X | A2] ),
  quicksort2( Big, A2-Z2).
```

Figure 9.3 A more efficient implementation of **quicksort** using difference-pair representation for lists. Relation **split(X, List, Small, Big)** is as defined in Figure 9.2.

The empty list is represented by any pair Z-Z. Introducing these changes systematically into the program of Figure 9.2 we get a more efficient implementation of quicksort, programmed as **quicksort2** in Figure 9.3. The procedure **quicksort** still uses the usual representation of lists, but the actual sorting is done by the more efficient **quicksort2**, which uses the difference-pair representation. The relation between the two procedures is:

```
quicksort( L, S) :-
  quicksort2( L, S-[] ).
```

Exercises

9.5 Write a procedure to merge two sorted lists producing a third list. For example:

```
?- merge( [2,5,6,6,8], [1,3,5,9], L).
L = [1,2,3,5,5,6,6,8,9]
```

9.6 The difference between the sorting programs of Figures 9.2 and 9.3 is in the representation of lists. The former uses plain lists whereas the latter uses difference-pair representation. Transformation between these two representations is straightforward and could be mechanized. Carry out the corresponding changes systematically in the program of Figure 9.2 to transform it into the program of Figure 9.3.

9.7 Our **quicksort** program performs badly when the list to be sorted is already sorted or almost sorted. Analyze why.

9.8 Another good idea for sorting a list that avoids the weakness of **quicksort** is based on dividing the list, then sorting smaller lists, and then merging

these sorted smaller lists. Accordingly, to sort a list L:

- divide L into two lists, L1 and L2, of approximately equal length;
- sort L1 and L2 giving S1 and S2;
- merge S1 and S2 giving L sorted.

Implement this sorting principle and compare its efficiency with the **quicksort** program.

9.2 Representing sets by binary trees

One usual application of lists is to represent sets of objects. A disadvantage of using the list for representing a set is that the set membership testing is relatively inefficient. The predicate **member(X, L)** to test whether X is a member of a list L is usually programmed as:

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

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

To find X in a list L, this procedure scans the list element by element until X is found or the end of the list is encountered. This is very inefficient in the case of long lists.

For representing sets, there are various tree structures that facilitate more efficient implementation of the set membership relation. We will here consider binary trees.

A binary tree is either empty or it consists of three things:

- a root
- a left subtree
- a right subtree

The root can be anything, but the subtrees have to be binary trees again. Figure 9.4 shows an example. This tree represents the set {a, b, c, d}. The elements of the set are stored as nodes of the tree. In Figure 9.4, the empty subtrees are not pictured; for example, the node b has two subtrees that are both empty.

There are many ways to represent a binary tree by a Prolog term. One simple possibility is to make the root of a binary tree the principal functor of the term, and the subtrees its arguments. Accordingly, the example tree of Figure 9.4 would be represented by:

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

Among other disadvantages, this representation requires another functor for

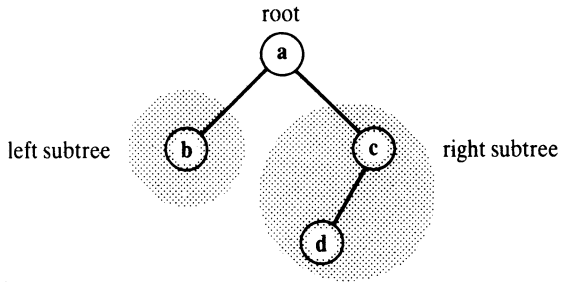


Figure 9.4 A binary tree.

each node of the tree. This can lead to troubles if nodes themselves are structured objects.

A better and more usual way to represent binary trees is as follows: we need a special symbol to represent the empty tree, and we need a functor to construct a non-empty tree from its three components (the root and the two subtrees). We will make the following choice regarding the functor and the special symbol:

- Let the atom **nil** represent the empty tree.
- Let the functor be **t** so the tree that has a root X, a left subtree L and a right subtree R is represented by the term **t(L, X, R)** (see Figure 9.5).

In this representation, the example tree of Figure 9.4 is represented by the term:

```
t( t( nil, b, nil), a, t( t( nil, d, nil), c, nil) )
```

Let us now consider the set membership relation, here named **in**. A goal

```
in( X, T)
```

is true if X is a node in a tree T. The **in** relation can be defined by the following rules:

X is in a tree T if:

- **the root of T is X, or**
- **X is in the left subtree of T, or**
- **X is in the right subtree of T.**

These rules directly translate into Prolog:

```
in( X, t( _, X, _) ).
```

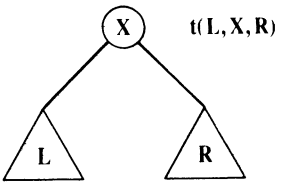


Figure 9.5 A representation of binary trees.

```
in( X, t( L, _, _) ) :-
  in( X, L).
in( X, t( _, _, R) ) :-
  in( X, R).
```

Obviously, the goal

```
in( X, nil)
```

will fail for any X.

Let us investigate the behaviour of this procedure. In the following examples, T is the tree of Figure 9.4. The goal

```
in( X, T)
```

will, through backtracking, find all the data in the set in the following order:

```
X = a;    X = b;    X = c;    X = d
```

Now let us consider efficiency. The goal

```
in( a, T)
```

succeeds immediately by the first clause of the procedure **in**. On the other hand, the goal

```
in( d, T)
```

will cause several recursive calls of **in** before d is eventually found. Similarly, the goal

```
in( e, T)
```

will fail only after the whole tree has been searched by recursive calls of **in** on *all* the subtrees of T.

This is, then, as inefficient as simply representing a set by a list. A major improvement can, however, be achieved if there is an ordering relation between the data in the set. Then the data in the tree can be ordered from left to right according to this relation. We say that a non-empty tree **t(Left, X, Right)** is ordered from left to right if:

(1) all the nodes in the left subtree, **Left**, are less than X; and
(2) all the nodes in the right subtree, **Right**, are greater than X; and
(3) both subtrees are also ordered.

Such a binary tree will be called a *binary dictionary*. Figure 9.6 shows an example.

The advantage of ordering is that, to search for an object in a binary dictionary, it is always sufficient to search at most one subtree. The key to this economization when searching for X is that we can by comparing X and the root immediately discard at least one of the subtrees. For example, let us search for the item 6 in the tree of Figure 9.6. We start at the root, 5, compare 6 and 5, and establish $6 > 5$. As all the data in the left subtree must be less than 5, the only remaining possibility to find 6 is the right subtree. So we continue the search in the right subtree, moving to node 8, etc.

The general method for searching in the binary dictionary is:

To find an item X in a dictionary D:

- if X is the root of D then X has been found, otherwise
- if X is less than the root of D then search for X in the left subtree of D, otherwise
- search for X in the right subtree of D;
- if D is empty the search fails.

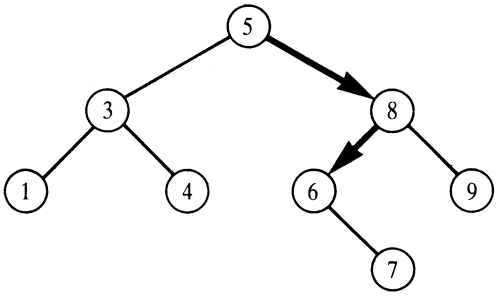


Figure 9.6 A binary dictionary. Item 6 is reached by following the indicated path $5 \rightarrow 8 \rightarrow 6$.

```
in( X, t( _, X, _) ).

in( X, t( Left, Root, Right) ) :-
  gt( Root, X),                    % Root greater than X
  in( X, Left).                    % Search left subtree

in( X, t( Left, Root, Right) ) :-
  gt( X, Root),                    % X greater than Root
  in( X, Right).                   % Search right subtree
```

Figure 9.7 Finding an item X in a binary dictionary.

These rules are programmed as the procedure **in** in Figure 9.7. The relation **gt(X, Y)** means: X is greater than Y. If the items stored in the tree are numbers then this relation is simply X > Y.

In a way, the **in** procedure itself can be also used for *constructing* a binary dictionary. For example, the following sequence of goals will construct a dictionary D that contains the elements 5, 3, 8:

```
?- in( 5, D), in( 3, D), in( 8, D).

D = t( t( D1, 3, D2), 5, t( D3, 8, D4) ).
```

The variables D1, D2, D3 and D4 are in fact four unspecified subtrees. They can be anything and D will still contain the given items 3, 5 and 8. The dictionary that is constructed depends on the order of goals in the question (Figure 9.8).

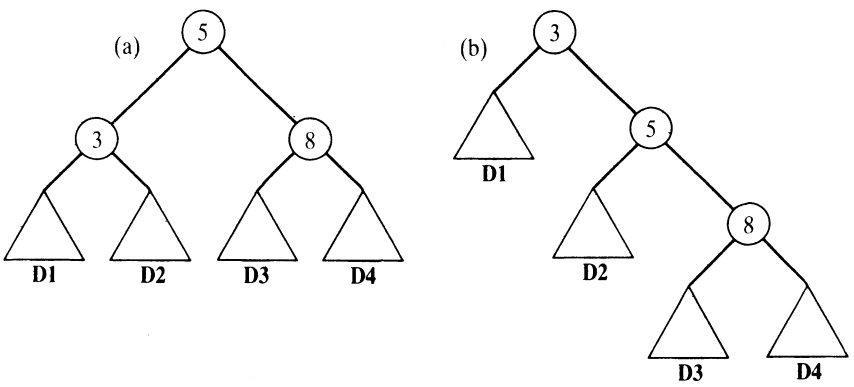


Figure 9.8 (a) Tree D that results from the sequence of goals; **in(5, D), in (3, D), in(8, D)**. (b) Tree resulting from: **in(3, D), in(5, D), in(8, D)**.

A comment is in order here on the search efficiency in dictionaries. Generally speaking, the search for an item in a dictionary is more efficient than searching in a list. What is the improvement? Let n be the number of items in our data set. If the set is represented by a list then the expected search time will be proportional to its length n. On average, we have to scan the list up to something like half-way through it. If the set is represented by a binary dictionary, the search time will be roughly proportional to the height of the tree. The height of a tree is the length of a longest path between the root and a leaf in the tree. The height, however, depends on the shape of the tree.

We say that a tree is (approximately) *balanced* if, for each node in the tree, its two subtrees accommodate approximately equal number of items. If a dictionary with n nodes is nicely balanced then its height is proportional to $\log n$. We say that a balanced tree has the *logarithmic complexity*. The difference between n and $\log n$ is the improvement of a balanced dictionary over a list. This holds, unfortunately, only when a tree is approximately balanced. If the tree gets out of balance its performance will degrade. In extreme cases of totally unbalanced trees, a tree is in effect reduced to a list. In such a case the tree's height is n, and the tree's performance is equally poor as that of a list. Therefore we are always interested in balanced dictionaries. Methods of achieving this objective will be discussed in Chapter 10.

Exercises

9.9 Define the predicates

binarytree(Object)
dictionary(Object)

to recognize whether **Object** is a binary tree or a binary dictionary respectively, written in the notation of this section.

9.10 Define the procedure

height(BinaryTree, Height)

to compute the height of a binary tree. Assume that the height of the empty tree is 0, and that of a one-element tree is 1.

9.11 Define the relation

linearize(Tree, List)

to collect all the nodes in **Tree** into a list.

9.12 Define the relation

maxelement(D, Item)

so that **Item** is the largest element stored in the binary dictionary D.

9.13 Modify the procedure

in(Item, BinaryDictionary)

by adding the third argument, **Path**, so that **Path** is the path between the root of the dictionary and **Item**.

9.3 Insertion and deletion in binary dictionary

When maintaining a dynamic set of data we may want to insert new items into the set and also delete some old items from the set. So one common repertoire of operations on a set of data, S, is:

in(X, S)	X is a member of S
add(S, X, S1)	Add X to S giving S1
del(S, X, S1)	Delete X from S giving S1

Let us now define the *add* relation. It is easiest to insert new data at the bottom level of the tree, so that a new item becomes a leaf of the tree at such a position that the ordering of the tree is preserved. Figure 9.9 shows changes in a tree during a sequence of insertions. Let us call this kind of insertion

addleaf(D, X, D1)

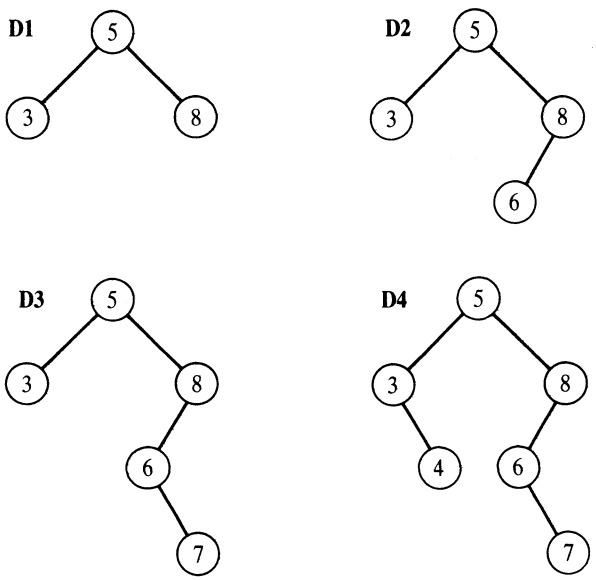


Figure 9.9 Insertion into a binary dictionary at the leaf level. The trees correspond to the following sequence of insertions: **add(D1, 6, D2), add(D2, 7, D3), add(D3, 4, D4).**

```
addleaf( nil, X, t( nil, X, nil) ).

addleaf( t( Left, X, Right), X, t( Left, X, Right) ).

addleaf( t( Left, Root, Right), X, t( Left1, Root, Right) ) :-
  gt( Root, X),
  addleaf( Left, X, Left1).

addleaf( t( Left, Root, Right), X, t( Left, Root, Right1) ) :-
  gt( X, Root),
  addleaf( Right, X, Right1).
```

Figure 9.10 Inserting an item as a leaf into the binary dictionary.

Rules for adding at the leaf level are:

- The result of adding X to the empty tree is the tree **t(nil, X, nil)**.
- If X is the root of D then D1 = D (no duplicate item gets inserted).
- If the root of D is greater than X then insert X into the left subtree of D; if the root of D is less than X then insert X into the right subtree.

Figure 9.10 shows a corresponding program.

Let us now consider the *delete* operation. It is easy to delete a leaf, but deleting an internal node is more complicated. The deletion of a leaf can be in fact defined as the inverse operation of inserting at the leaf level:

```
delleaf( D1, X, D2) :-
  addleaf( D2, X, D1).
```

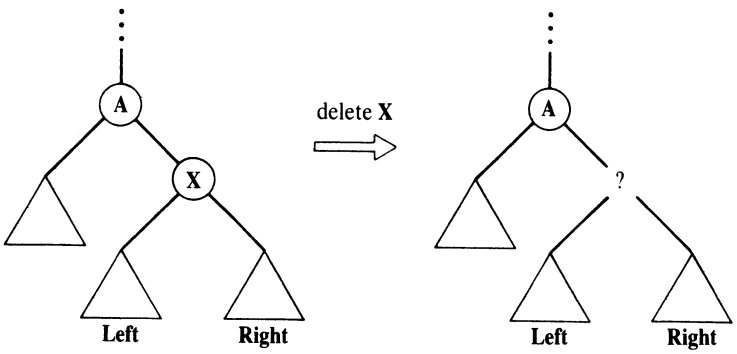


Figure 9.11 Deleting **X** from a binary dictionary. The problem is how to patch up the tree after **X** is removed.

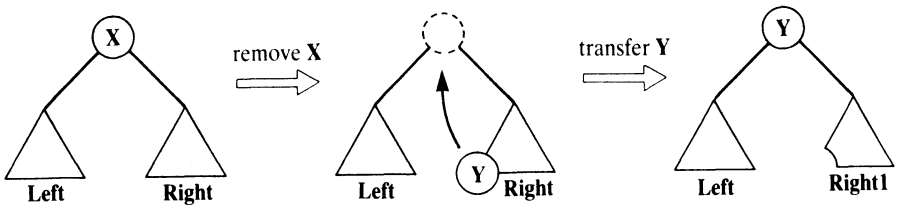


Figure 9.12 Filling the gap after removal of **X**.

Unfortunately, if X is an internal node then this does not work because of the problem illustrated in Figure 9.11. X has two subtrees, **Left** and **Right**. After X is removed, we have a hole in the tree and **Left** and **Right** are no longer connected to the rest of the tree. They cannot both be directly connected to the father of X, A, because A can accommodate only one of them.

If one of the subtrees **Left** and **Right** is empty then the solution is simple: the non-empty subtree is connected to A. If they are both non-empty then one idea is as shown in Figure 9.12. The left-most node of **Right**, Y, is transferred from its current position upwards to fill the gap after X. After this transfer, the tree remains ordered. Of course, the same idea works symmetrically, with the transfer of the right-most node of **Left**.

According to these considerations, the operation to delete an item from the binary dictionary is programmed in Figure 9.13. The transfer of the left-most node of the right subtree is accomplished by the relation

delmin(Tree, Y, Tree1)

```
del( t( nil, X, Right), X, Right).

del( t( Left, X, nil), X, Left).

del( t( Left, X, Right), X, t( Left, Y, Right1) ) :-
  delmin( Right, Y, Right1).

del( t( Left, Root, Right), X, t( Left1, Root, Right) ) :-
  gt( Root, X),
  del( Left, X, Left1).

del( t( Left, Root, Right), X, t( Left, Root, Right1) ) :-
  gt( X, Root),
  del( Right, X, Right1).

delmin( t( nil, Y, R), Y, R).

delmin( t( Left, Root, Right), Y, t( Left1, Root, Right) ) :-
  delmin( Left, Y, Left1).
```

Figure 9.13 Deleting from the binary dictionary.

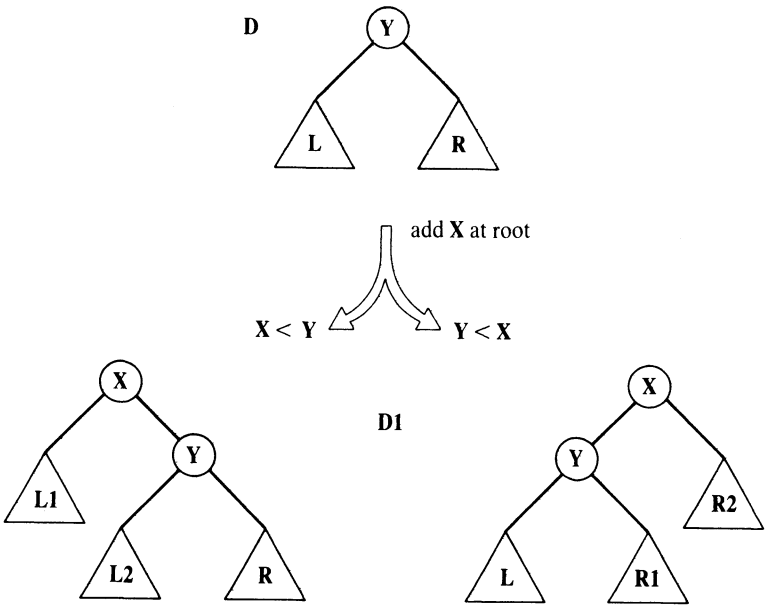


Figure 9.14 Inserting **X** at the root of a binary dictionary.

where Y is the minimal (that is, the left-most) node of **Tree**, and **Tree1** is **Tree** with Y deleted.

There is another elegant solution to *add* and *delete*. The *add* relation can be defined non-deterministically so that a new item is inserted at any level of the tree, not just at the leaf level. The rules are:

To add X to a binary dictionary D either:

- add X at the root of D (so that X becomes the new root), or
- if the root of D is greater than X then insert X into the left subtree of D, otherwise insert X into the right subtree of D.

The difficult part of this is the insertion at the root of D. Let us formulate this operation as a relation

addroot(D, X, D1)

where X is the item to be inserted at the root of D and D1 is the resulting dictionary with X as its root. Figure 9.14 illustrates the relations between X, D and D1. The remaining question is now: What are the subtrees L1 and L2 in Figure 9.14 (or R1 and R2 alternatively)? The answer can be derived from the following constraints:

- L1 and L2 must be binary dictionaries;

- the set of nodes in L1 and L2 is equal to the set of nodes in L;
- all the nodes in L1 are less than X, and all the nodes in L2 are greater than X.

The relation that imposes all these constraints is just our **addroot** relation. Namely, if X were added as the root into L, then the subtrees of the resulting tree would be just L1 and L2. In Prolog: L1 and L2 must satisfy the goal

```
addroot( L, X, t( L1, X, L2) )
```

The same constraints apply to R1 and R2:

```
addroot( R, X, t( R1, X, R2) )
```

Figure 9.15 shows a complete program for the 'non-deterministic' insertion into the binary dictionary.

The nice thing about this insertion procedure is that there is no restriction on the level of insertion. Therefore *add* can be used in the inverse direction in order to delete an item from the dictionary. For example, the following goal list

```
add( D, X, D1) :-
  addroot( D, X, D1).                    % Add X as new root

add( t( L, Y, R), X, t( L1, Y, R) ) :-   % Insert X into left subtree
  gt( Y, X),
  add( L, X, L1).

add( t( L, Y, R), X, t( L, Y, R1) ) :-   % Insert X into right subtree
  gt( X, Y),
  add( R, X, R1).

addroot( nil, X, t( nil, X, nil) ).      % Insert into empty tree

addroot( t( L, X, R), X, t( L, X, R) ).  % X already in tree

addroot( t( L, Y, R), X, t( L1, X, t( L2, Y, R) ) ) :-
  gt( Y, X),
  addroot( L, X, t( L1, X, L2) ).

addroot( t( L, Y, R), X, t( t( L, Y, R1), X, R2) ) :-
  gt( X, Y),
  addroot( R, X, t( R1, X, R2) ).
```

Figure 9.15 Insertion into the binary dictionary at any level of the tree.

constructs a dictionary D containing the items 3, 5, 1, 6, and then deletes 5 yielding a dictionary DD:

```
add( nil, 3, D1), add( D1, 5, D2), add( D2, 1, D3),
add( D3, 6, D), add( DD, 5, D)
```

9.4 Displaying trees

Like all data objects in Prolog, a binary tree, T, can be directly output by the built-in procedure **write**. However, the goal

```
write( T)
```

will only output all the information, but will not graphically indicate the actual tree structure. It can be rather tiring to imagine the actual tree structure from a Prolog term that represents that tree. Therefore it is often desirable to have a tree typed out in a way that graphically indicates its structure.

There is a relatively simple method for displaying trees in such a form. The trick is to display a tree growing from left to right, and not from top to bottom as trees are usually pictured. The tree is rotated to the left so that the root becomes the left-most element, and the leaves are moved to the right. Figure 9.16 illustrates.

Let us define a procedure

```
show( T)
```

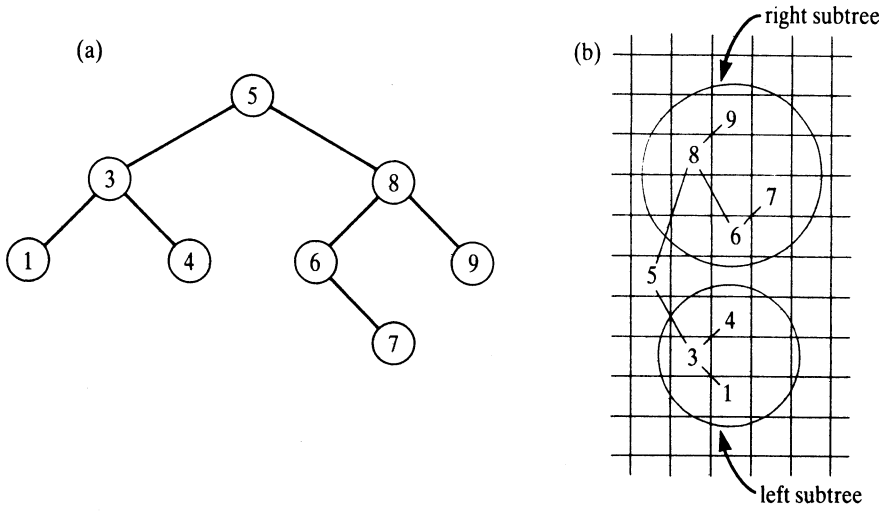


Figure 9.16 (a) A tree as normally pictured. (b) The same tree as typed out by the procedure **show** (arcs are added for clarity).

```
show( T) :-
  show2( T, 0).

show2( nil, _).

show2( t( L, X, R), Indent) :-
  Ind2 is Indent + 2,
  show2( R, Ind2),
  tab( Indent), write( X), nl,
  show2( L, Ind2).
```

Figure 9.17 Displaying a binary tree.

to display a tree T in the form indicated in Figure 9.16. The principle is:

To show a non-empty tree, T:

(1) show the right subtree of T, indented by some distance, H, to the right;
(2) write the root of T;
(3) show the right subtree of T indented by distance H to the right.

The indentation distance H, which can be appropriately chosen, is an additional parameter for displaying trees. Introducing H we have the procedure

show2(T, H)

to display T indented H spaces from the left margin. The relation between the procedures **show** and **show2** is:

show(T) :- show2(T, 0).

The complete program, which indents by 2, is shown in Figure 9.17. The principle of achieving such an output format can be easily adopted for displaying other types of trees.

Exercise

9.14 Our procedure for displaying trees shows a tree in an unusual orientation, so that the root is on the left and the leaves of the tree are on the right. Write a (more difficult) procedure to display a tree in the usual orientation with the root at the top and the leaves at the bottom.

9.5 Graphs

9.5.1 Representing graphs

Graph structures are used in many applications, such as representing relations, situations or problems. A graph is defined by a set of *nodes* and a set of *edges*, where each edge is a pair of nodes. When the edges are directed they are also called *arcs*. Arcs are represented by *ordered* pairs. Such a graph is a *directed* graph. The edges can be attached costs, names, or any kind of labels, depending on the application. Figure 9.18 shows examples.

Graphs can be represented in Prolog in several ways. One method is to represent each edge or arc separately as one clause. The graphs in Figure 9.18 can be thus represented by sets of clauses, for example:

```
connected( a, b).
connected( b, c).
...

arc( s, t, 3).
arc( t, v, 1).
arc( u, t, 2).
...
```

Another method is to represent a whole graph as one data object. A graph can be thus represented as a pair of two sets: nodes and edges. Each set can be represented as a list; each edge is a pair of nodes. Let us choose the functor **graph** to combine both sets into a pair, and the functor **e** for edges. Then one way to represent the (undirected) graph in Figure 9.18 is:

```
G1 = graph( [a,b,c,d], [e(a,b), e(b,d), e(b,c), e(c,d)] )
```

To represent a directed graph we can choose the functors **digraph** and **a** (for arcs). The directed graph of Figure 9.18 is then:

```
G2 = digraph( [s,t,u,v], [a(s,t,3), a(t,v,1), a(t,u,5), a(u,t,2), a(v,u,2)] )
```

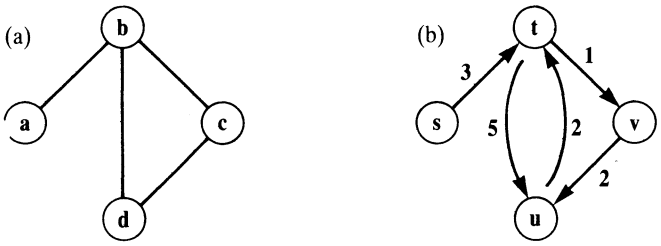


Figure 9.18 (a) A graph. (b) A directed graph with costs attached to the arcs.

If each node is connected to at least one other node then we can omit the list of nodes from the representation as the set of nodes is then implicitly specified by the list of edges.

Yet another method is to associate with each node a list of nodes that are adjacent to that node. Then a graph is a list of pairs consisting of a node plus its adjacency list. Our example graphs can then, for example, be represented by:

```
G1 = [ a -> [b], b -> [a,c,d], c -> [b,d], d -> [b,c] ]

G2 = [ s -> [t/3], t -> [u/5, v/1], u -> [t/2], v -> [u/2] ]
```

The symbols '->' and '/' above are, of course, infix operators.

What will be the most suitable representation will depend on the application and on operations to be performed on graphs. Two typical operations are:

- find a path between two given nodes;
- find a subgraph, with some specified properties, of a graph.

Finding a spanning tree of a graph is an example of the latter operation. In the following sections we will look at some simple programs for finding a path and for finding a spanning tree.

9.5.2 Finding a path

Let G be a graph, and A and Z two nodes in G. Let us define a relation

```
path( A, Z, G, P)
```

where P is an acyclic path between A and Z in G. P is represented as a list of nodes on the path. If G is the graph in the left-hand side of Figure 9.18 then:

```
path( a, d, G, [a,b,d] )

path( a, d, G, [a,b,c,d] )
```

Since a path must not contain any cycle, a node can appear in the path at most once. One method to find a path is:

```
To find an acyclic path, P, between A and Z in a graph, G:

If A = Z then P = [], otherwise
find an acyclic path, P1, from some node Y to Z, and find
a path from A to Y avoiding the nodes in P1.
```

This formulation implies another relation: find a path under the restriction of

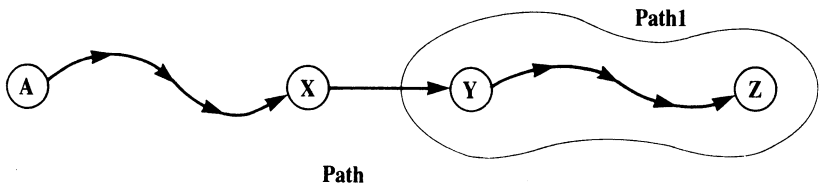


Figure 9.19 The **path1** relation: **Path** is a path between A and Z; the last part of **Path** overlaps with **Path1**.

avoiding some subset of nodes (P1 above). We will, accordingly, define another procedure:

path1(A, P1, G, P)

As illustrated in Figure 9.19, the arguments are:

- A is a node,
- G is a graph,
- P1 is a path in G,
- P is an acyclic path in G that goes from A to the beginning of P1 and continues along P1 up to its end.

The relation between **path** and **path1** is:

path(A, Z, G, P) :- path1(A, [Z], G, P).

Figure 9.19 suggests a recursive definition of **path1**. The boundary case arises when the start node of P1 (Y in Figure 9.19) coincides with the start node of P, A. If the start nodes do not coincide then there must be a node, X, such that:

(1) Y is adjacent to X, and
(2) X is not in P1, and
(3) P must satisfy the relation **path1(A, [X | P1], G, P)**

A complete program is shown in Figure 9.20. In this program, **member** is the list membership relation. The relation

adjacent(X, Y, G)

means that there is an arc from X to Y in graph G. The definition of this relation depends on the representation of graphs. If G is represented as a pair of sets (nodes and edges),

G = graph(Nodes, Edges)

```
path( A, Z, Graph, Path) :-
  path1( A, [Z], Graph, Path).

path1( [A | Path1], _, [A | Path1] ).

path1( A, [Y | Path1], Graph, Path) :-
  adjacent( X, Y, Graph),
  not member( X, Path1),                    % No-cycle condition
  path1( A, [X, Y | Path1], Graph, Path).
```

Figure 9.20 Finding an acyclic path, **Path**, from A to Z in **Graph**.

then:

```
adjacent( X, Y, graph( Nodes, Edges) ) :-
  member( e(X,Y), Edges);
  member( e(Y,X), Edges).
```

A classical problem on graphs is to find a Hamiltonian path; that is, an acyclic path comprising all the nodes in the graph. Using **path** this can be done as follows:

```
hamiltonian( Graph, Path) :-
  path( _, _, Graph, Path),
  covers( Path, Graph).

covers( Path, Graph) :-
  not ( node( N, Graph), not member( N, Path) ).
```

Here, **node(N, Graph)** means: N is a node in **Graph**.

We can attach costs to paths. The cost of a path is the sum of the costs of the arcs in the path. If there are no costs attached to the arcs then we can talk about the length instead, counting 1 for each arc in the path. Our **path** and **path1** relations can be modified to handle costs by introducing an additional argument, the cost, for each path:

```
path( A, Z, G, P, C)

path1( A, P1, C1, G, P, C)
```

Here, C is the cost of P and C1 is the cost of P1. The relation **adjacent** now also has an extra argument, the cost of an arc. Figure 9.21 shows a path-finding program that computes a path and its cost.

This procedure can be used for finding a minimum cost path. We can find

```
path( A, Z, Graph, Path, Cost) :-
  path1( A, [Z], 0, Graph, Path, Cost).

path1( A, [A | Path1], Cost1, Graph, [A | Path1], Cost1).

path1( A, [Y | Path1], Cost1, Graph, Path, Cost) :-
  adjacent( X, Y, CostXY, Graph),
  not member( X, Path1),
  Cost2 is Cost1 + CostXY,
  path1( A, [X, Y | Path1], Cost2, Graph, Path, Cost).
```

Figure 9.21 Path-finding in a graph: **Path** is an acyclic path with cost **Cost** from A to Z in **Graph**.

such a path between two nodes, **node1** and **node2**, in some graph G by the goals:

```
path( node1, node2, Graph, MinPath, MinCost),
not ( path( node1, node2, Graph, _, Cost), Cost < MinCost) )
```

We can also find a maximum cost path between any pair of nodes in a graph G by the goals:

```
path( _, _, Graph, MaxPath, MaxCost),
not ( path( _, _, Graph, _, Cost), Cost > MaxCost) )
```

It should be noted that this is a very inefficient way for finding minimal or maximal paths. This method unselectively investigates possible paths and is completely unsuitable for large graphs because of its high time complexity. The path-finding problem frequently arises in Artificial Intelligence. We will study more sophisticated methods for finding optimal paths in Chapters 11 and 12.

9.5.3 Finding a spanning tree of a graph

A graph is said to be *connected* if there is a path from any node to any other node. Let G = (V, E) be a connected graph with the set of nodes V and the set of edges E. A *spanning tree* of G is a connected graph T = (V, E′) where E′ is a subset of E such that:

(1) T is connected, and
(2) there is no cycle in T.

These two conditions guarantee that T is a tree. For the left-hand side graph of

Figure 9.18, there are three spanning trees, which correspond to three lists of edges:

Tree1 = [a-b, b-c, c-d]

Tree2 = [a-b, b-d, d-c]

Tree3 = [a-b, b-d, b-c]

Here each term of the form X-Y denotes an edge between nodes X and Y. We can pick any node in such a list as the root of a tree. Spanning trees are of interest, for example, in communication problems because they provide, with the minimum number of communication lines, a path between any pair of nodes.

We will define a procedure

stree(T, G)

where T is a spanning tree of G. We will assume that G is connected. We can imagine constructing a spanning tree algorithmically as follows: Start with the empty set of edges and gradually add new edges from G, taking care that a cycle is never created, until no more edge can be added because it would create a cycle. The resulting set of edges defines a spanning tree. The no-cycle condition can be maintained by a simple rule: an edge can be added only if one of its nodes is already in the growing tree, and the other node is not yet in the tree. A program that implements this idea is shown in Figure 9.22. The key relation in this program is

spread(Tree1, Tree, G)

All the three arguments are sets of edges. G is a connected graph; **Tree1** and **Tree** are subsets of G such that they both represent trees. **Tree** is a spanning tree of G obtained by adding zero or more edges of G to **Tree1**. We can say that '**Tree1** gets spread to **Tree**'.

It is interesting that we can also develop a working program for constructing a spanning tree in another, completely declarative way, by simply stating mathematical definitions. We will assume that both graphs and trees are represented by lists of their edges, as in the program of Figure 9.22. The definitions we need are:

(1) T is a spanning tree of G if:

- T is a subset of G, and
- T is a tree, and
- T 'covers' G; that is, each node of G is also in T.

```
% Finding a spanning tree of a graph
%
% Trees and graphs are represented by lists of their
% edges. For example:
%        Graph = [a-b, b-c, b-d, c-d]

stree( Graph, Tree) :-                    % Tree is a spanning tree of Graph
  member( Edge, Graph),
  spread( [Edge], Tree, Graph).

spread( Tree1, Tree, Graph) :-
  addedge( Tree1, Tree2, Graph),
  spread( Tree2, Tree, Graph).

spread( Tree, Tree, Graph) :-
  not addedge( Tree, _, Graph).           % No edge can be added
                                          % without creating a cycle
addedge( Tree, [A-B | Tree], Graph) :-
  adjacent( A, B, Graph),                 % Nodes A and B adjacent in Graph
  node( A, Tree),                         % A in Tree
  not node( B, Tree).                     % A-B doesn't create a cycle in Tree

adjacent( A, B, Graph) :-
  member( A-B, Graph);
  member( B-A, Graph).

node( A, Graph) :-                        % A is a node in Graph if
  adjacent( A, _, Graph).                 % A is adjacent to anything in Graph
```

Figure 9.22 Finding a spanning tree of a graph: an 'algorithmic' program. The program assumes that the graph is connected.

(2) A set of edges T is a tree if:

- T is connected, and
- T has no cycle.

Using our **path** program of the previous section, these definitions can be stated in Prolog as shown in Figure 9.23. It should be noted, however, that this program is, in this form, of little practical interest because of its inefficiency.

Exercise

9.15 Consider spanning trees of graphs that have costs attached to edges. Let the *cost* of a spanning tree be defined as the sum of the costs of all the edges in the tree. Write a program to find a minimum-cost spanning tree of a given graph.

```
% Finding a spanning tree
% Graphs and trees are represented as lists of edges.

stree( Graph, Tree) :-
  subset( Graph, Tree),
  tree( Tree),
  covers( Tree, Graph).

tree( Tree) :-
  connected( Tree),
  not hasacycle( Tree).

connected( Tree) :-
  not ( node( A, Tree), node( B, Tree), not path( A, B, Tree, _) ).

hasacycle( Tree) :-
  adjacent( A, B, Tree),
  path( A, B, Tree, [A, X, Y | _] ).          % A path of length > 1

covers( Tree, Graph) :-
  not ( node( A, Graph), not node( A, Tree) ).

subset( [], [] ).

subset( [X | L], S) :-
  subset( L, L1),
  ( S = L1; S = [X | L1] ).
```

Figure 9.23 Finding a spanning tree of a graph: a 'declarative' program. Relations **node** and **adjacent** are as in Figure 9.22.

Summary

In this chapter we studied Prolog implementations of some frequently used data structures and associated operations on them. These include:

- Lists:

  variations in representing lists

  sorting lists:

  bubble sort
  insertion sort
  quicksort
  efficiency of these procedures

- Representing sets as binary trees and binary dictionaries:

  searching for an item in a tree
  adding an item
  deleting an item
  adding as a leaf, adding as the root
  the balance of trees and its relation to the efficiency of these operations
  displaying trees

- Graphs:

  representing graphs
  finding a path in a graph
  finding a spanning tree of a graph

References

In this chapter we have tackled important topics of sorting and of maintaining data structures for representing sets. General description of structures and algorithms that were programmed in this chapter can be found, for example, in Aho, Hopcroft and Ullman (1974, 1984) or Baase (1978). The behaviour of these algorithms, in particular their time complexity, is also studied. Gonnet (1983) is a good and concise collection of many related algorithms and results of their mathematical analysis.

The Prolog program for insertion at any level of the binary tree (Section 9.3) was first shown to the author by M. van Emden (personal communication).

Aho, A. V., Hopcroft, J. E. and Ullman, J. D. (1974) *The Design and Analysis of Computer Algorithms*. Addison-Wesley.

Aho, A. V., Hopcroft, J. E. and Ullman, J. D. (1983) *Data Structures and Algorithms*. Addison-Wesley.

Baase, S. (1978) *Computer Algorithms*. Addison-Wesley.

Gonnet, G. H. (1984) *Handbook of Algorithms and Data Structures*. Addison-Wesley.

10 Advanced Tree Representations

In this chapter we look at advanced techniques for representing data sets by trees. The key idea is to keep the tree balanced, or approximately balanced, in order to prevent the tree from degenerating toward a list. Such tree-balancing schemes guarantee relatively fast, logarithmic-time data-access even in the worst case. Two such schemes are presented in this chapter: 2-3 trees and AVL-trees. (The knowledge of this chapter is not a prerequisite to any other chapter.)

10.1 The 2-3 dictionary

A binary tree is said to be well balanced if both its subtrees are of approximately equal height (or size) and they are also balanced. The height of a balanced tree is approximately $\log n$ where n is the number of nodes in the tree. The time needed to evaluate the relations **in**, **add** and **delete** on binary dictionaries grows proportionally with the height of the tree. On balanced dictionaries, then, all these operations can be done in time that is in the order of $\log n$. The logarithmic growth of the complexity of the set membership testing is a definite improvement over the list representation of sets, where the complexity grows linearly with the size of the data set. However, poor balance of a tree will degrade the performance of the dictionary. In extreme cases, the binary dictionary degenerates into a list, as shown in Figure 10.1. The form of the

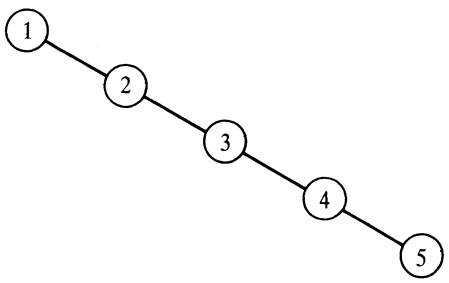


Figure 10.1 A totally unbalanced binary dictionary. Its performance is reduced to that of a list.

dictionary depends on the sequence in which the data is inserted. In the best case we get a good balance with performance in the order log n, and in the worst case the performance is in the order n. Analysis shows that on average, assuming that any sequence of data is equally likely, the complexity of **in**, **add** and **delete** is still in the order log n. So the average performance is, fortunately, closer to the best case than to the worst case. There are, however, several rather simple schemes for keeping good balance of the tree regardless of the data sequence. Such schemes guarantee the *worst case* performance of **in**, **add** and **delete** in the order log n. One of them is the 2-3 tree; another scheme is the AVL-tree.

The *2-3* tree is defined as follows: it is either empty, or it consists of a single node, or it is a tree that satisfies the following conditions:

- each internal node has two or three children, and
- all the leaves are at the same level.

A 2-3 dictionary is a 2-3 tree in which the data items are stored in the leaves, ordered from left to right. Figure 10.2 shows an example. The internal nodes contain labels that specify the minimal elements of the subtrees as follows:

- if an internal node has two subtrees, this internal node contains the minimal element of the second subtree;
- if an internal node has three subtrees then this node contains the minimal elements of the second and of the third subtree.

To search for an item, X, in a 2-3 dictionary we start at the root and move toward the bottom level according to the labels in the internal nodes. Let the root contain the labels M1 and M2. Then:

- if X < M1 then continue the search in the left subtree, otherwise

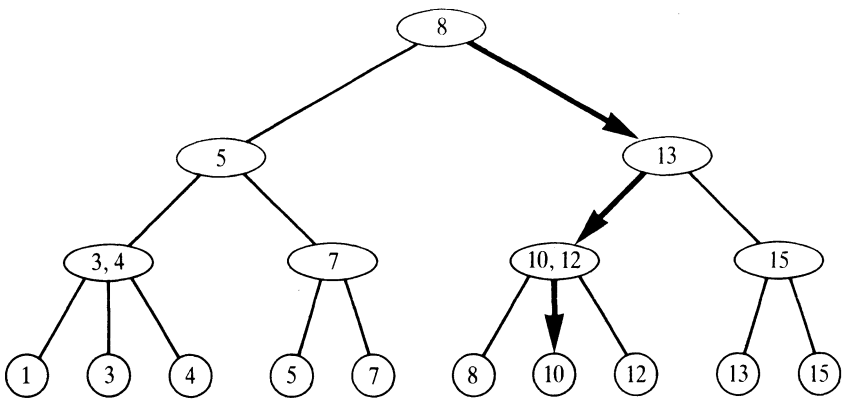


Figure 10.2 A 2-3 dictionary. The indicated path corresponds to searching for the item 10.

- if X < M2 then continue the search in the middle subtree, otherwise
- continue the search in the right subtree.

If the root only contains one label, M, then proceed to the left subtree if X < M, and to the right subtree otherwise. This is repeated until the leaf level is reached, and at this point X is either successfully found or the search fails.

As all the leaves are at the same level, the 2-3 tree is perfectly balanced with respect to the heights of the subtrees. All search paths from the root to a leaf are of the same length which is of the order log n, where n is the number of items stored in the tree.

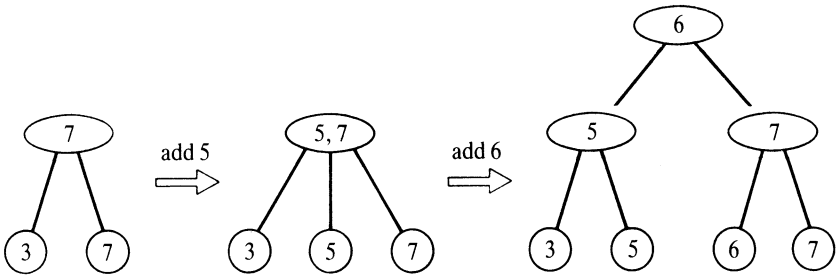


Figure 10.3 Inserting into a 2-3 dictionary. The tree first grows in breadth and then upwards.

When inserting new data, the 2-3 tree can also grow in breadth, not only in depth. Each internal node that has two children can accommodate an additional child which results in the breadth-wise growth. If, on the other hand, a node with three children accepts another child then this node is split into two nodes, each of them taking over two of the total of four children. The so-generated new internal node gets incorporated further up in the tree. If this happens at the top level then the tree is forced to grow upwards. Figure 10.3 illustrates these principles.

Insertion into the 2-3 dictionary will be programmed as a relation

add23(Tree, X, NewTree)

where **NewTree** is obtained by inserting X into **Tree**. The main burden of insertion will be transferred to two auxiliary relations, both called **ins**. The first one has three arguments:

ins(Tree, X, NewTree)

where **NewTree** is the result of inserting X into **Tree**. **Tree** and **NewTree** have the *same height*. But, of course, it is not always possible to preserve the same

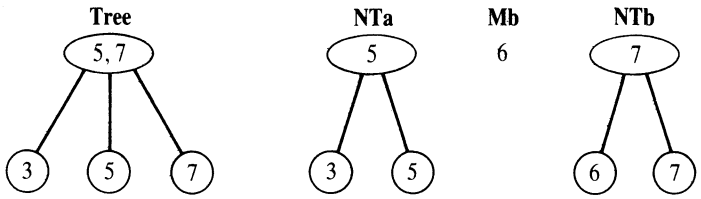


Figure 10.4 The objects in the figure satisfy the relation **ins(Tree, 6, NTa, Mb, NTb)**.

height after insertion. Therefore we have another **ins** relation, with five arguments, to cater for this case:

```
ins( Tree, X, NTa, Mb, NTb)
```

Here, when inserting X into **Tree**, **Tree** is split into two trees: **NTa** and **NTb**. Both **NTa** and **NTb** have the same height as **Tree**. **Mb** is the minimal element of **NTb**. Figure 10.4 shows an example.

In the program, a 2-3 tree will be represented, depending on its form, as follows:

- **nil** represents the empty tree
- **l(X)** represents a single node tree, a leaf with item X
- **n2(T1, M, T2)** represents a tree with two subtrees, T1 and T2; M is the minimal element of T2
- **n3(T1, M2, T2, M3, T3)** represents a tree with three subtrees, T1, T2 and T3; M2 is the minimal element of T2, and M3 is the minimal element of T3

T1, T2 and T3 are all 2-3 trees.

The relation between **add23** and **ins** is: if after insertion the tree does not grow upwards then simply

```
add23( Tree, X, NewTree) :-
  ins( Tree, X, NewTree).
```

If, however, the height after insertion increases, then **ins** determines the two subtrees, T1 and T2, which are then combined into a bigger tree:

```
add23( Tree, X, n2( T1, M, T2) ) :-
  ins( Tree, X, T1, M, T2).
```

The **ins** relation is more complicated because it has to deal with many cases: inserting into the empty tree, a single node tree, a tree of type n2 or n3. Additional subcases arise from insertion into the first, second or third subtree.

Accordingly, **ins** will be defined by a set of rules so that each clause about **ins** will deal with one of the cases. Figure 10.5 illustrates some of these cases. The cases in this figure translate into Prolog as follows:

Case a

```
ins( n2( T1, M, T2), X, n2( NT1, M, T2) ) :-
  gt( M, X),                                  % M greater than X
  ins( T1, X, NT1).
```

Case b

```
ins( n2( T1, M, T2), X, n3( NT1a, Mb, NT1b, M, T2) ) :-
  gt( M, X),
  ins( T1, X, NT1a, Mb, NT1b).
```

Case c

```
ins( n3( T1, M2, T2, M3, T3), X,
     n2( NT1a, Mb, NT1b), M2, n2( T2, M3, T3) ) :-
  gt( M2, X),
  ins( T1, X, NT1a, Mb, NT1b).
```

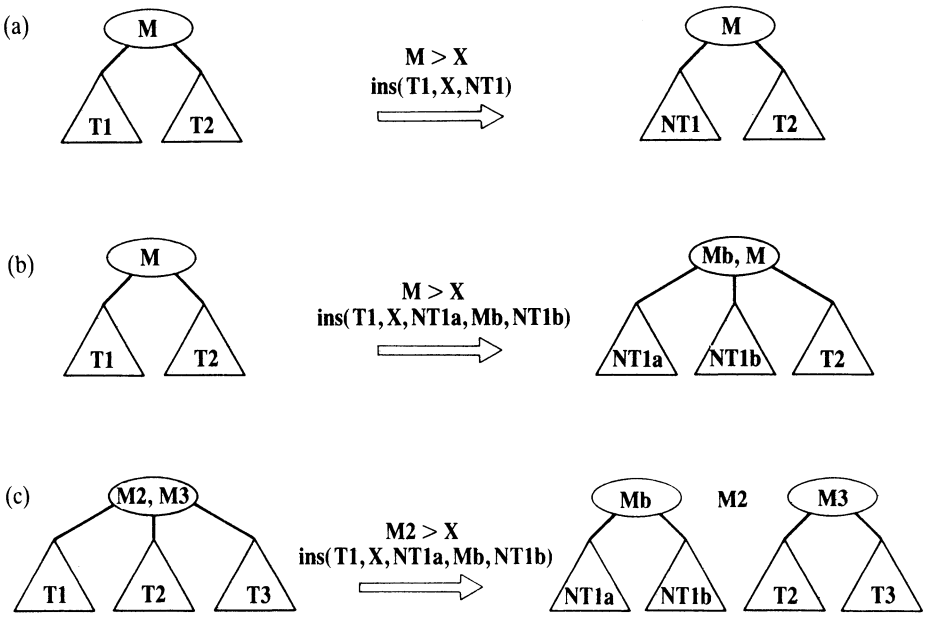


Figure 10.5 Some cases of the **ins** relation.
(a) **ins(n2(T1, M, T2), X, n2(NT1, M, T2));**
(b) **ins(n2(T1, M, T2), X, n3(NT1a, Mb, NT1b, M, T2));**
(c) **ins(n3(T1, M2, T2, M3, T3), X, n2(NT1a, Mb, NT1b), M2, n2(T2, M3, T3)).**

```
% Insertion and deletion in the 2-3 dictionary

add23( Tree, X, Tree1) :-                    % Add X to Tree giving Tree1
  ins( Tree, X, Tree1).                      % Tree grows in breadth

add23( Tree, X, n2( T1, M2, T2) ) :-         % Tree grows upwards
  ins( Tree, X, T1, M2, T2).

del23( Tree, X, Tree1) :-                    % Delete X from Tree giving Tree1
  add23( Tree1, X, Tree).

ins( nil, X, l(X) ).

ins( l(A), X, l(A), X, l(X) ) :-
  gt( X, A).

ins( l(A), X, l(X), A, l(A) ) :-
  gt( A, X).

ins( n2( T1, M, T2), X, n2( NT1, M, T2) ) :-
  gt( M, X),
  ins( T1, X, NT1).

ins( n2( T1, M, T2), X, n3( NT1a, Mb, NT1b, M, T2) ) :-
  gt( M, X),
  ins( T1, X, NT1a, Mb, NT1b).

ins( n2( T1, M, T2), X, n2( T1, M, NT2) ) :-
  gt( X, M),
  ins( T2, X, NT2).

ins( n2( T1, M, T2), X, n3( T1, M, NT2a, Mb, NT2b) ) :-
  gt( X, M),
  ins( T2, X, NT2a, Mb, NT2b).

ins( n3( T1, M2, T2, M3, T3), X, n3( NT1, M2, T2, M3, T3) ) :-
  gt( M2, X),
  ins( T1, X, NT1).

ins( n3( T1, M2, T2, M3, T3), X,
     n2( NT1a, Mb, NT1b), M2, n2( T2, M3, T3) ) :-
  gt( M2, X),
  ins( T1, X, NT1a, Mb, NT1b).

ins( n3( T1, M2, T2, M3, T3), X, n3( T1, M2, NT2, M3, T3) ) :-
  gt( X, M2), gt( M3, X),
  ins( T2, X, NT2).
```

```
ins( n3( T1, M2, T2, M3, T3), X,
     n2( T1, M2, NT2a), Mb, n2( NT2b, M3, T3) ) :-
  gt( X, M2), gt( M3, X),
  ins( T2, X, NT2a, Mb, NT2b).

ins( n3( T1, M2, T2, M3, T3), X, n3( T1, M2, T2, M3, NT3) ) :-
  gt( X, M3),
  ins( T3, X, NT3).

ins( n3( T1, M2, T2, M3, T3), X,
     n2( T1, M2, T2), M3, n2( NT3a, Mb, NT3b) ) :-
  gt( X, M3),
  ins( T3, X, NT3a, Mb, NT3b).
```

Figure 10.6 Inserting and deleting in the 2-3 dictionary. In this program, an attempt to insert a duplicate item will fail.

Figure 10.6 shows the complete program for inserting into and deleting from the 2-3 dictionary. The **add23** procedure can be used in the reverse direction for deleting by:

```
del23( T, X, T1) :- add23( T1, X, T).
```

Figure 10.7 shows a program for displaying 2-3 trees.

Our program occasionally does some unnecessary backtracking. If the three-argument **ins** fails then the five-argument **ins** is called, which redoes part of the work. This source of inefficiency can easily be eliminated by, for example, redefining **ins** as:

```
ins2( Tree, X, NewTrees)
```

NewTrees is a list of length 1 or 3, as follows:

```
NewTrees = [ NewTree] if ins( Tree, X, NewTree)
NewTrees = [ NTa, Mb, NTb] if ins( Tree, X, NTa, Mb, NTb)
```

The **add23** relation would be, accordingly, redefined as:

```
add23( T, X, T1) :-
  ins2( T, X, Trees),
  combine( Trees, T1).
```

The **combine** relation has to produce a single tree, T1, from the list **Trees**.

```
% Displaying 2-3 dictionary

show( T) :-
  show( T, 0).

show( nil, _).

show( l(A), H) :-
  tab( H), write( A), nl.

show( n2( T1, M, T2), H) :-
  H1 is H + 5,
  show( T2, H1),
  tab( H), write( --), nl,
  tab( H), write( M), nl,
  tab( H), write( --), nl,
  show( T1, H1).

show( n3( T1, M2, T2, M3, T3), H) :-
  H1 is H + 5,
  show( T3, H1),
  tab( H), write( --), nl,
  tab( H), write( M3), nl,
  show( T2, H1),
  tab( H), write( M2), nl,
  tab( H), write( --), nl,
  show( T1, H1).
```

(a)

```
                15
                --
                15
                --
                13
        --
        13
        --
                12
                --
                12
                10
                10
                --
                8
--
8
--
                7
                --
                7
                --
                5
        --
        5
        --
                4
                --
                4
                3
                3
                --
                1
```

(b)

Figure 10.7 (a) A program to display a 2-3 dictionary. (b) The dictionary of Figure 10.2 as displayed by this program.

Exercises

10.1 Define the relation

in(Item, Tree)

to search for **Item** in a 2-3 dictionary **Tree**.

10.2 Modify the program of Figure 10.6 to avoid backtracking (define relations **ins2** and **combine**).

10.2 AVL-tree: an approximately balanced tree

AVL-tree is a binary tree that has the following properties:

(1) Its left subtree and right subtree differ in height by 1 at the most.
(2) Both subtrees themselves are also AVL-trees.

This definition allows for trees that are slightly out of balance. It can be shown that the height of an AVL-tree is always, even in the worst case, roughly proportional to log n where n is the number of nodes in the tree. This guarantees the logarithmic performance for the operations **in**, **add** and **del**.

Operations on the AVL-dictionary are essentially the same as on binary dictionaries, with some additions to maintain approximate balance of the tree. If the tree gets out of approximate balance after an insertion or deletion then some additional mechanism will get it back into the required degree of balance. To implement this mechanism efficiently, we have to maintain some additional information about the balance of the tree. Essentially we only need the difference between the heights of its subtrees, which is either -1, 0 or $+1$. For the sake of simplicity of the operations involved we will, however, prefer to maintain the complete heights of trees and not only the differences.

We will define the insertion relation as

addavl(Tree, X, NewTree)

where both **Tree** and **NewTree** are AVL-dictionaries such that **NewTree** is **Tree** with X inserted. AVL-trees will be represented by terms of the form:

t(Left, A, Right)/Height

where A is the root, **Left** and **Right** are the subtrees, and **Height** is the height of the tree. The empty tree is represented by **nil/0**. Now let us consider the insertion of X into a non-empty AVL-dictionary

Tree = t(L, A, R)/H

We will start our discussion by only considering the case that X is greater than A. Then X is to be inserted into R and we have the following relation:

addavl(R, X, t(R1, B, R2)/Hb)

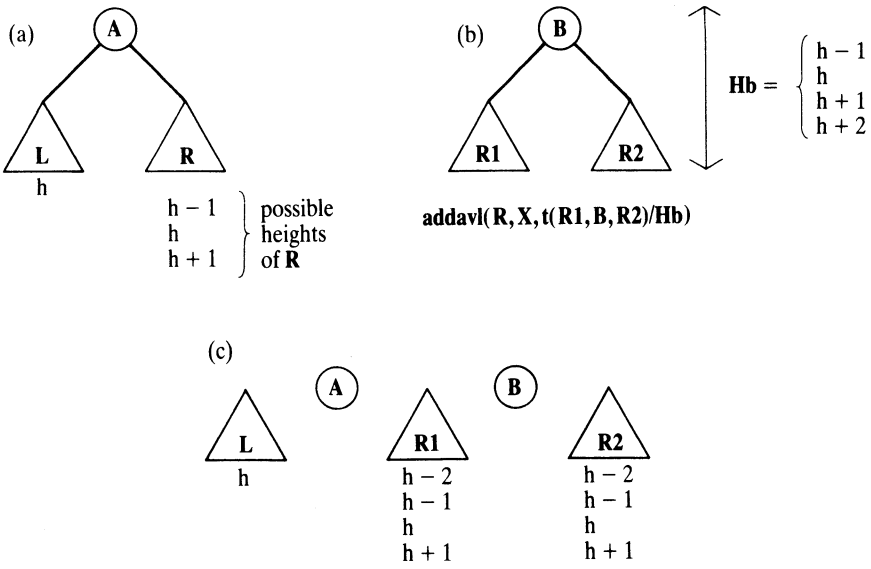


Figure 10.8 The problem of AVL insertion: (a) AVL-tree before inserting **X**, **X** > **A**; (b) AVL-tree after inserting **X** into **R**; (c) ingredients from which the new tree is to be constructed.

Figure 10.8 illustrates the following ingredients from which **NewTree** is to be constructed:

L, A, R1, B, R2

What can be the heights of L, R, R1 and R2? L and R can only differ in height by 1 at the most. Figure 10.8 shows what the heights of R1 and R2 can be. As only one item, X, has been inserted into R, at most one of the subtrees R1 and R2 can have the height $h + 1$.

In the case that X is less than A then the situation is analogous with left and right subtrees interchanged. Therefore, in any case, we have to construct **NewTree** from three trees (let us call them **Tree1**, **Tree2** and **Tree3**), and two single items, A and B. Let us now consider the question: How can we combine these five ingredients to make **NewTree** so that **NewTree** is an AVL-dictionary? The order from left to right in **NewTree** has to be:

Tree1, A, Tree2, B, Tree3

We have to consider three cases:

(1) The middle tree, **Tree2**, is taller than both other trees.

(2) **Tree1** is at least as tall as **Tree2** and **Tree3**.

(3) **Tree3** is at least as tall as **Tree2** and **Tree1**.

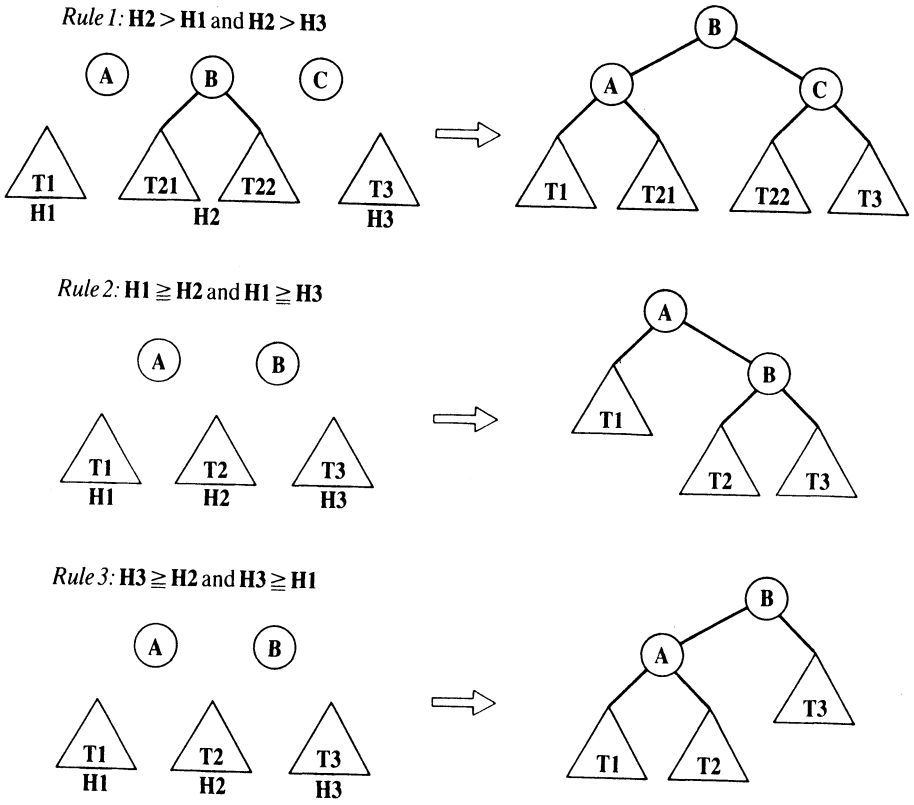


Figure 10.9 Three combination rules for AVL-trees.

Figure 10.9 shows how **NewTree** can be constructed in each of these cases. In case 1, the middle tree **Tree2** has to be decomposed and its parts incorporated into **NewTree**. The three rules of Figure 10.9 are easily translated into Prolog as a relation:

```
combine( Tree1, A, Tree2, B, Tree3, NewTree)
```

The last argument, **NewTree**, is an AVL-tree constructed from five ingredients, the first five arguments. Rule 1, for example, becomes:

```
combine(
  T1/H1, A, t(T21,B,T22)/H2, C, T3/H3,          % Five ingredients
  t( t(T1/H1,A,T21)/Ha, B, t(T22,C,T3/H3)/Hc)/Hb)  :- % Their combination
  H2 > H1, H2 > H1,                  % Middle tree is tallest
  Ha is H1 + 1,                      % Height of left subtree
  Hc is H3 + 1,                      % Height of right subtree
  Hb is Ha + 1.                      % Height of the whole tree
```

```
% Insertion into AVL-dictionary

addavl( nil/0, X, t( nil/0, X, nil/0)/1).          % Add X to empty tree

addavl( t( L, Y, R)/Hy, X, NewTree) :-             % Add X to non-empty tree
  gt( Y, X),
  addavl( L, X, t( L1, Z, L2)/_ ),                 % Add into left subtree
  combine( L1, Z, L2, Y, R, NewTree).              % Combine ingredients of NewTree

addavl( t( L, Y, R)/Hy, X, NewTree) :-
  gt( X, Y),
  addavl( R, X, t( R1, Z, R2)/_ ),                 % Add into right subtree
  combine( L, Y, R1, Z, R2, NewTree).

combine( T1/H1, A, t( T21, B, T22)/H2, C, T3/H3,
         t( t(T1/H1,A,T21)/Ha, B, t(T22,C,T3/H3)/Hc)/Hb) :-
  H2 > H1, H2 > H3,                                % Middle subtree tallest
  Ha is H1 + 1,
  Hc is H3 + 1,
  Hb is Ha + 1.

combine( T1/H1, A, T2/H2, C, T3/H3,
         t( T1/H1, A, t(T2/H2,C,T3/H3)/Hc)/Ha) :-
  H1 >= H2, H1 >= H3,                              % Tall left subtree
  max1( H2, H3, Hc),
  max1( H1, Hc, Ha).

combine( T1/H1, A, T2/H2, C, T3/H3,
         t( t(T1/H1,A,T2/H2)/Ha, C, T3/H3)/Hc) :-
  H3 >= H2, H3 >= H1,                              % Tall right subtree
  max1( H1, H2, Ha),
  max1( Ha, H3, Hc).

max1( U, V, M) :-                                  % M is 1 + max. of U and V
  U > V, !, M is U + 1;
  M is V + 1.
```

Figure 10.10 AVL-dictionary insertion. In this program, an attempt to insert a duplicate will fail. See Figure 10.9 for **combine**.

A complete **addavl** program, which also computes the heights of the tree and the subtrees, is shown as Figure 10.10.

Our program works with the heights of trees. A more economical representation is, as said earlier, possible. In fact, we only need the balance, which can only be –1, 0 or +1. The disadvantage of such economization would be, however, somewhat more complicated combination rules.

Exercise

10.3 Define the relation

avl(Tree)

to test whether a binary tree **Tree** is an AVL-tree; that is, all the sibling subtrees may differ in their heights by 1 at the most. Let binary trees be represented by terms of the form **t(Left, Root, Right)** or **nil**.

Summary

- *2-3 trees* and *AVL-trees*, implemented in this chapter, are types of *balanced* trees.
- Balanced, or approximately balanced, trees guarantee efficient execution of the three basic operations on trees: looking for an item, adding or deleting an item. All these operations can be done in time proportional to log n, where n is the number of nodes in the tree.

References

2-3 trees are described in detail by, for example, Aho, Hopcroft and Ullman (1974, 1983). In their 1983 book an implementation in Pascal is also given. Wirth (1976) gives a Pascal program to handle AVL-trees. 2-3 trees are a special case of more general B-trees. This and several other variations or data structures related to 2-3 trees and AVL-trees are covered by Gonnet (1984) together with various results on the behaviour of these structures.

A program for AVL-tree insertion that only uses tree-bias information (that is, the difference between the heights of the subtrees -1, 0 or +1, and not the complete height) was published by van Emden (1981).

Aho, A. V., Hopcroft, J. E. and Ullman, J. D. (1974) *The Design and Analysis of Computer Algorithms*. Addison-Wesley.

Aho, A. V., Hopcroft, J. E. and Ullman, J. D. (1983) *Data Structures and Algorithms*. Addison-Wesley.

Gonnet, G. H. (1984) *Handbook of Algorithms and Data Structures*. Addison-Wesley.

van Emden, M. (1981) *Logic Programming Newsletter* **2**.

Wirth, N. (1976) *Algorithms and Data Structures = Programs*. Prentice-Hall.

11 Basic Problem-Solving Strategies

This chapter is centred around a general scheme, called *state space*, for representing problems. A state space is a graph whose nodes correspond to problem situations, and a given problem is reduced to finding a path in this graph. We will study examples of formulating problems using the state-space approach, and discuss general methods for solving problems represented in this formalism. Problem-solving involves graph searching and typically leads to the problem of dealing with alternatives. Two basic strategies for exploring alternatives, presented in this chapter, are the depth-first search and the breadth-first search.

11.1 Introductory concepts and examples

Let us consider the example in Figure 11.1. The problem is to find a plan for rearranging a stack of blocks as shown in the figure. We are only allowed to move one block at a time. A block can be grasped only when its top is clear. A block can be put on the table or on some other block. To find a required plan, we have to find a sequence of moves that accomplish the given transformation.

We can think of this problem as a problem of exploring among possible alternatives. In the initial problem situation we are only allowed one alternative: put block C on the table. After C has been put on the table, we have three alternatives:

- put A on table, or
- put A on C, or
- put C on A

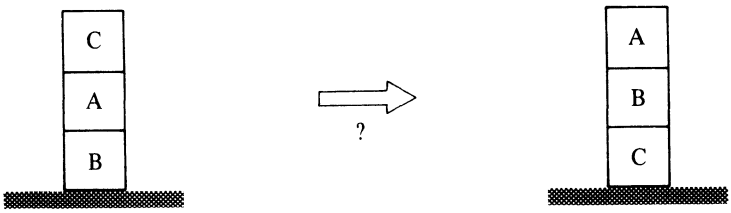


Figure 11.1 A blocks rearrangement problem.

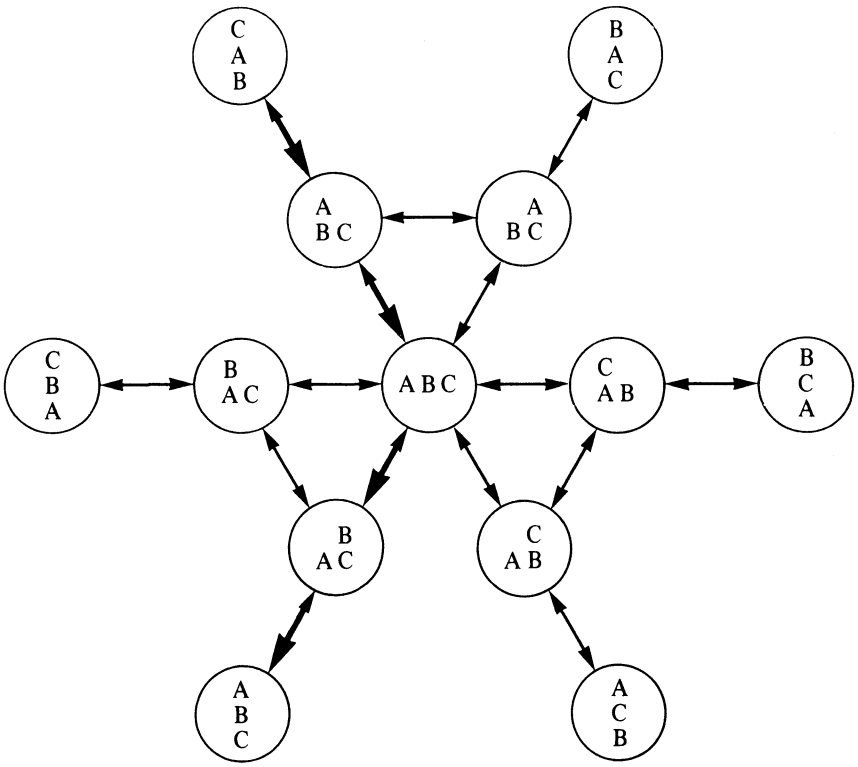


Figure 11.2 A graphical representation of the block manipulation problem. The indicated path is a solution to the problem in Figure 11.1.

We will not seriously consider putting C on the table as this clearly has no effect on the situation.

As this example illustrates, we have, in such a problem, two types of concept:

(1) Problem situations.

(2) Legal moves, or actions, that transform problem situations into other situations.

Problem situations and possible moves form a directed graph, called a *state space*. A state space for our example problem is shown in Figure 11.2. The nodes of the graph correspond to problem situations, and the arcs correspond to legal transitions between states. The problem of finding a solution plan is equivalent to finding a path between the given initial situation (the start node) and some specified final situation, also called a *goal node*.

Figure 11.3 shows another example problem: an eight puzzle and its representation as a path-finding problem. The puzzle consists of eight sliding tiles, numbered by digits from 1 to 8, and arranged in a 3 by 3 array of nine cells.

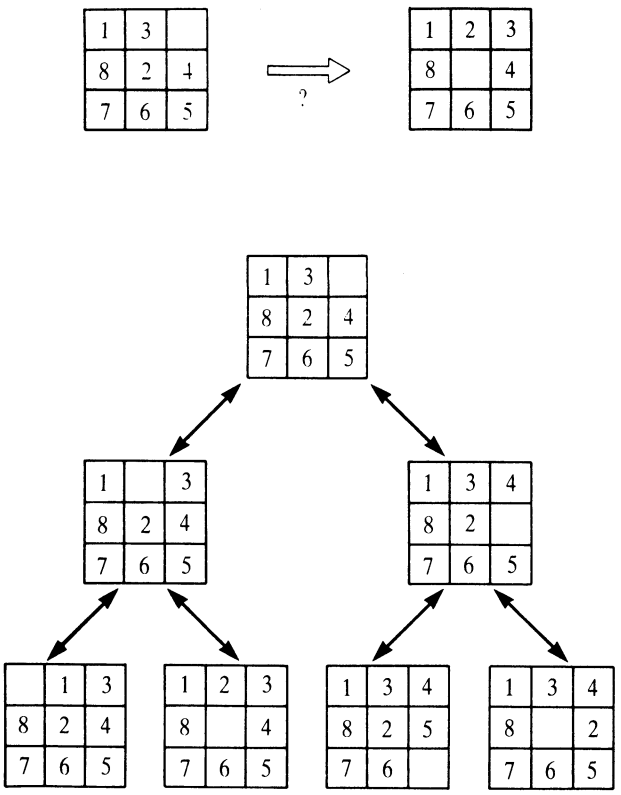


Figure 11.3 An eight puzzle and a corresponding graphical representation.

One of the cells is always empty, and any adjacent tile can be moved into the empty cell. We can say that the empty cell is allowed to move around, swapping its place with any of the adjacent tiles. The final situation is some special arrangement of tiles, as shown for example in Figure 11.3.

It is easy to construct similar graph representations for other popular puzzles. Straightforward examples are the Tower of Hanoi, or getting fox, goose and grain across the river. In the latter problem, the boat can only hold the farmer and one other object, and the farmer has to protect the goose from the fox, and the grain from the goose. Many practical problems also naturally fit this paradigm. Among them is the travelling salesman problem which is the formal model of many practical optimization problems. The problem is defined by a map with n cities and road distances between the cities. The task is to find a shortest route from some starting city, visiting all the cities and ending in the starting city. No city, with the exception of the starting one, may appear in the tour twice.

Let us summarize the concepts introduced by these examples. The state space of a given problem specifies the 'rules of the game': nodes in the state space correspond to situations, and arcs correspond to 'legal moves', or

actions, or solution steps. A particular problem is defined by:

- a state space,
- a start node,
- a goal condition (a condition to be reached); 'goal nodes' are those nodes that satisfy this condition.

We can attach costs to legal moves or actions. For example, costs attached to moving blocks in the block manipulation problem would indicate that some blocks are harder to move than others. In the travelling salesman problem, moves correspond to direct city-to-city journeys. Naturally, the costs of such moves are the distances between the cities.

In cases where costs are attached to moves, we are normally interested in minimum cost solutions. The cost of a solution is the sum of the costs of the arcs along the solution path. Even if no costs are given we may have an optimization problem: we may be interested in shortest solutions.

Before presenting some programs that implement classical algorithms for searching state spaces, let us first discuss how a state space can be represented in a Prolog program.

We will represent a state space by a relation

s(X, Y)

which is true if there is a legal move in the state space from a node X to a node Y. We will say that Y is a *successor* of X. If there are costs associated with moves then we will add a third argument, the cost of the move:

s(X, Y, Cost)

This relation can be represented in the program explicitly by a set of facts. For typical state spaces of any significant complexity this would be, however, impractical or impossible. Therefore the successor relation, s, is usually defined implicitly by stating the rules for computing successor nodes of a given node.

Another question of general importance is, how to represent problem situations, that is nodes themselves. The representation should be compact, but it should also enable efficient execution of operations required; in particular, the evaluation of the successor relation, and possibly the associated costs.

As an example, let us consider the block manipulation problem of Figure 11.1. We will consider a more general case, so that there are altogether any number of blocks that are arranged in one or more stacks. The number of stacks will be limited to some given maximum to make the problem more interesting. This may also be a realistic constraint because a robot that manipulates blocks may be only given a limited working space on the table.

A problem situation can be represented as a list of stacks. Each stack can

be, in turn, represented by a list of blocks in that stack ordered so that the top block in the stack is the head of the list. Empty stacks are represented by empty lists. The initial situation of the problem in Figure 11.1 can be thus represented by:

```
[ [c,a,b], [], [] ]
```

A goal situation is any arrangement with the ordered stack of all the blocks. There are three such situations:

```
[ [a,b,c], [], [] ]

[ [], [a,b,c], [] ]

[ [], [], [a,b,c] ]
```

The successor relation can be programmed according to the following rule: **Situation2** is a successor of **Situation1** if there are two stacks, **Stack1** and **Stack2**, in **Situation1**, and the top block of **Stack1** can be moved to **Stack2**. As all situations are represented as lists of stacks, this is translated into Prolog as:

```
s( Stacks, [Stack1, [Top1 | Stack2] | OtherStacks] ) :- %'Move Top1 to Stack2
   del( [Top1 | Stack1], Stacks, Stacks1),              % Find first stack
   del( Stack2, Stacks1, OtherStacks).                 % Find second stack

del( X, [X | L], L).

del( X, [Y | L], [Y | L1] ) :-
  del( X, L, L1).
```

The goal condition for our example problem is:

```
goal( Situation) :-
  member( [a,b,c],  Situation).
```

We will program search algorithms as a relation

```
solve( Start, Solution)
```

where **Start** is the start node in the state space, and **Solution** is a path between **Start** and any goal node. For our block manipulation problem the corresponding call can be:

```
?- solve( [ [c,a,b], [], [] ], Solution).
```

As the result of a successful search, **Solution** is instantiated to a list of block arrangements. This list represents a plan for transforming the initial state into a state in which all the three blocks are in one stack arranged as **[a,b,c]**.

11.2 Depth-first search strategy

Given a state-space formulation of a problem, there are many approaches to finding a solution path. Two basic search strategies are: *depth-first* search and *breadth-first* search. In this section we will implement the first of them.

We will start the development of this algorithm and its variations with a simple idea:

To find a solution path, Sol, from a given node, N, to some goal node:

- **if N is a goal node then Sol = [N], or**
- **if there is a successor node, N1, of N, such that there is a path Sol1 from N1 to a goal node, then Sol = [N | Sol1].**

This translates into Prolog as:

```
solve( N, [N] ) :-
  goal( N).

solve( N, [N1 | Sol1] ) :-
  s( N, N1),
  solve( N1, Sol1).
```

This program is in fact an implementation of the depth-first strategy. It is called 'depth-first' because of the order in which the alternatives in the state space are explored. Whenever the depth-first algorithm is given a choice of continuing the search from several nodes it always decides to choose a deepest one. A 'deepest' node is one that is farthest from the start node. Figure 11.4 illustrates

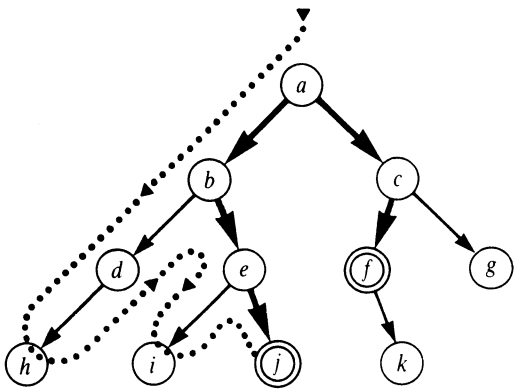


Figure 11.4 A simple state space: *a* is the start node, *f* and *j* are goal nodes. The order in which the depth-first strategy visits the nodes in this state space is: *a*, *b*, *d*, *h*, *e*, *i*, *j*. The solution found is: **[a,b,e,j]**. On backtracking, the other solution is discovered: **[a,c,f]**.

the order in which the nodes are visited. This order corresponds to tracing the way that Prolog answers the question:

```
?- solve( a, Sol).
```

The depth-first search is most amenable to the recursive style of programming in Prolog. The reason for this is that Prolog itself, when executing goals, explores alternatives in the depth-first fashion.

The depth-first search is simple and easy to program, and may work well in certain cases. The eight queens programs of Chapter 4 were, in fact, examples of depth-first search. A state-space formulation of the eight queens problem that could be used by the **solve** procedure above can be as follows:

- nodes are board positions with zero or more queens placed in consecutive files of the board;
- a successor node is obtained by placing another queen into the next file so that she does not attack any of the existing queens;
- the start node is the empty board represented by the empty list;
- a goal node is any position with eight queens (the successor rule guarantees that the queens do not attack each other).

Representing the board position as a list of Y-coordinates of the queens, this can be programmed as:

```
s( Queens, [Queen | Queens] ) :-
   member( Queen, [1,2,3,4,5,6,7,8] ),      % Place Queen into any row
   noattack( Queen, Queens).

goal( [_, _, _, _, _, _, _, _] ).           % Position with 8 queens
```

The **noattack** relation requires that **Queen** does not attack any of the **Queens**; it can be easily programmed as in Chapter 4. The question

```
?- solve( [], Solution).
```

will produce a list of board positions with increasing number of queens. The list will end with a safe configuration of eight queens. It will also find alternative solutions through backtracking.

The depth-first search often works well, as in this example, but there are many ways in which our simple **solve** procedure can run into trouble. Whether this will actually happen or not depends on the state space. To embarass our **solve** procedure with the problem of Figure 11.4, a slight modification of this problem is sufficient: add an arc from *h* to *d*, thus creating a cycle (Figure 11.5). The search would in this case proceed as follows: start at *a* and descend to *h* following the left-most branch of the graph. At this point, in contrast with Figure 11.4, *h* has a successor, *d*. Therefore the execution will *not backtrack*

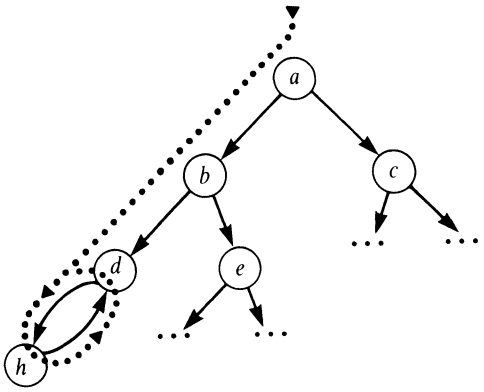


Figure 11.5 Starting at a, the depth-first search ends in cycling between d and h: $a, b, d, h, d, h, d \ldots$.

from h, but *proceed* to d instead. Then the successor of d, h, will be found, etc., resulting in cycling between d and h.

An obvious improvement of our depth-first program is to add a cycle-detection mechanism. Accordingly, any node that is already in the path from the start node to the current node should not be considered again. We can formulate this as a relation

depthfirst(Path, Node, Solution)

As illustrated in Figure 11.6, **Node** is the state from which a path to a goal state is to be found; **Path** is a path (a list of nodes) between the start node and **Node**; **Solution** is **Path** extended via **Node** to a goal node.

For the sake of ease of programming, paths will be in our program represented by lists in the *inverse* order. The argument **Path** can be used for two purposes:

(1) to prevent the algorithm from considering those successors of **Node** that have already been encountered (cycle detection);

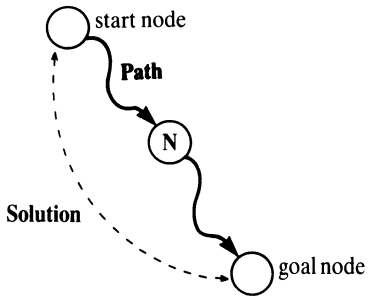


Figure 11.6 Relation **depthfirst(P, N, Solution)**.

```
solve( Node, Solution) :-
  depthfirst( [], Node, Solution).

depthfirst( Path, Node, [Node | Path] ) :-
  goal( Node).

depthfirst( Path, Node, Sol) :-
  s( Node, Node1),
  not member( Node1, Path),                    % Prevent a cycle
  depthfirst( [Node | Path], Node1, Sol).
```

Figure 11.7 A depth-first search program that avoids cycling.

(2) to construct a solution path **Solution**.

A corresponding depth-first search program is shown in Figure 11.7.

Let us outline a variation of this program. Two arguments of this procedure, **Path** and **Node**, can be combined into a list **[Node | Path]**. So, instead of a candidate node **Node**, which aspires to be on a path to a goal, we have a candidate *path* **P = [Node | Path]**, which aspires to be extendable up to a goal node. The programming of the corresponding predicate

depthfirst1(P, Solution)

is left as an exercise for the reader.

With the cycle-detection mechanism, our depth-first procedure will find solution paths in state spaces such as that in Figure 11.5. There are, however, state spaces in which this program will still easily get lost. Many state spaces are infinite. In such a space, the depth-first algorithm may miss a goal node, proceeding along an infinite branch of the graph. The program may then indefinitely explore this infinite part of the space never getting closer to a goal. The eight queens state space, as defined in this section, may seem to be susceptible to this kind of trap. However, this space is, incidentally, finite, because by the limited choice of Y-coordinates eight queens at most can be placed safely.

To avoid aimless infinite (non-cyclic) branches, we can add another refinement to the basic depth-first search procedure: limiting the depth of search. We then have the following arguments for the depth-first search procedure:

depthfirst2(Node, Solution, Maxdepth)

The search is not allowed to go in depth beyond **Maxdepth**. This constraint can be programmed by decreasing the depth limit at each recursive call, and not

```
depthfirst2( Node, [Node], _) :-
  goal( Node).

depthfirst2( Node, [Node | Sol], Maxdepth) :-
  Maxdepth > 0,
  s( Node, Node1),
  Max1 is Maxdepth - 1,
  depthfirst2( Node1, Sol, Max1).
```

Figure 11.8 A depth-limited, depth-first search program.

allowing this limit to become negative. The resulting program is shown in Figure 11.8.

Exercises

11.1 Write a depth-first search procedure (with cycle detection)

depthfirst1(CandidatePath, Solution)

to find a solution path **Solution** as an extension of **CandidatePath**. Let both paths be represented as lists of nodes in the inverse order, so that the goal node is the head of **Solution**.

11.2 Write a depth-first procedure that combines both the cycle-detection and the depth-limiting mechanisms of the procedures in Figures 11.7 and 11.8.

11.3 Experiment with the depth-first programs of this section in the blocks world planning problem of Figure 11.1.

11.4 Write a procedure

show(Situation)

to display a problem state, **Situation**, in the blocks world. Let **Situation** be a list of stacks, and a stack in turn a list of blocks. The goal

show([[a], [e,d], [c,b]])

should display the corresponding situation; for example, as:

```
      e   c
  a   d   b
  ===========
```

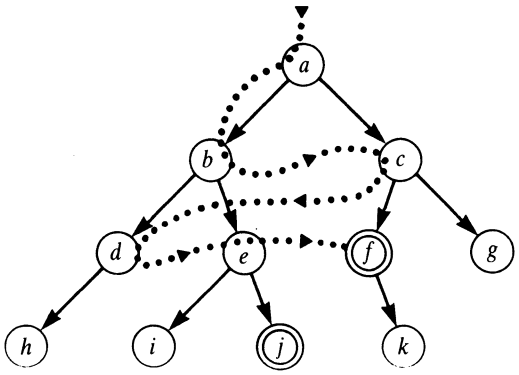


Figure 11.9 A simple state space: *a* is the start node, *f* and *j* are goal nodes. The order in which the breadth-first strategy visits the nodes in this state space is: *a*, *b*, *c*, *d*, *e*, *f*. The shorter solution **[a,c,f]** is found before the longer one **[a,b,e,j]**.

11.3 Breadth-first search strategy

In contrast to the depth-first search strategy, the breadth-first search strategy chooses to first visit those nodes that are closest to the start node. This results in a search process that tends to develop more into breadth than into depth, as illustrated by Figure 11.9.

The breadth-first search is not so easy to program as the depth-first search. The reason for this difficulty is that we have to maintain a *set* of alternative candidate nodes, not just one as in depth-first search. This set of candidates is the whole growing bottom edge of the search tree. However, even this set of nodes is not sufficient if we also want to extract a solution path from the search process. Therefore, instead of maintaining a set of candidate nodes, we maintain a set of candidate *paths*. Then,

breadthfirst(Paths, Solution)

is true if some path from a candidate set **Paths** can be extended to a goal node. **Solution** is such an extended path.

11.3.1 Representing the candidate set as a list

In our first implementation of this idea we will use the following representation for the set of candidate paths. The set will be represented as a list of paths, and each path will be a list of nodes in the inverse order; that is, the head will be the most recently generated node, and the last element of the list will be the start node of the search. The search is initiated with a single element candidate set:

[[StartNode]]

An outline for breadth-first search is:

To do the breadth-first search when given a set of candidate paths:

- if the first path contains a goal node as its head then this is a solution of the problem, otherwise
- remove the first path from the candidate set and generate the set of all possible one-step extensions of this path, adding this set of extensions at the end of the candidate set, and execute breadth-first search on this updated set.

For our example problem of Figure 11.9, this process develops as follows:

(1) Start with the initial candidate set:

[[a]]

(2) Generate extensions of **[a]**:

[[b,a], [c,a]]

(Note that all paths are represented in the inverse order.)

(3) Remove the first candidate path, **[b,a]**, from the set and generate extensions of this path:

[[d,b,a], [e,b,a]]

Add the list of extensions to the end of the candidate set:

[[c,a], [d,b,a], [e,b,a]]

(4) Remove **[c,a]** and add its extensions to the end of the candidate set, producing:

[[d,b,a], [e,b,a], [f,c,a], [g,c,a]]

In further steps, **[d,b,a]** and **[e,b,a]** are extended and the modified candidate set becomes:

[[f,c,a], [g,c,a], [h,d,b,a], [i,e,b,a], [j,e,b,a]]

Now the search process encounters **[f,c,a]** which contains a goal node, **f**. Therefore this path is returned as a solution.

A program that carries out this process is shown in Figure 11.10. In this program all one-step extensions are generated by using the built-in procedure **bagof**. A test to prevent the generation of cyclic paths is also made. Note that in the case that no extension is possible, **bagof** fails and therefore an alternative call to **breadthfirst** is provided. **member** and **conc** are the list membership and list concatenation relations respectively.

```
solve( Start, Solution) :-
  breadthfirst( [ [Start] ], Solution).

breadthfirst( [ [Node | Path] | _], [Node | Path] ) :- !.
  goal( Node).

breadthfirst( [ [N | Path] | Paths], Solution) :-
  bagof( [M, N | Path],
         ( s( N, M), not member( M, [N | Path] ) ),
         NewPaths),                    % NewPaths = acyclic extensions of [N | Path]
    conc( Paths, NewPaths, Paths1), !,
    breadthfirst( Paths1, Solution);
    breadthfirst( Paths, Solution).    % Case that N has no successor
```

Figure 11.10 An implementation of breadth-first search.

A drawback of this program is the inefficiency of the **conc** operation. This can be rectified by using the difference-pair representation of lists introduced in Chapter 8. The set of candidate paths would then be represented by a pair of lists, **Paths** and **Z**, written as:

```
Paths - Z
```

Introducing this representation into the program of Figure 11.10, it can be systematically transformed into the program shown in Figure 11.11. This transformation is left as an exercise for the reader.

```
solve( Start, Solution) :-
  bfirst( [ [Start] | Z]-Z, Solution).

bfirst( [ [Node | Path] | _]-_, [Node | Path] ) :-
  goal( Node).

bfirst( [ [N | Path] | Paths]-Z, Solution) :-
  bagof( [M, N | Path], ( s( N, M), not member( M, [N | Path] ) ), New),
    conc( New, ZZ, Z), !,
    bfirst( Paths-ZZ, Solution);
  Paths \== Z,                         % Set of candidate paths non-empty
    bfirst( Paths-Z, Solution).
```

Figure 11.11 A more efficient program than that of Figure 11.10 for the breadth-first search. The improvement is based on using the difference-pair representation for the list of candidate paths.

11.3.2 Using tree representation for the candidate set

Let us now consider another modification of our breadth-first program. In our implementation so far, the set of candidate paths was represented as a list of paths. This is wasteful because initial parts of the paths are shared among several paths. Thus these shared parts are stored repeatedly in several paths. A more compact representation of the set of candidate paths should avoid such redundancy. Such a more compact representation is a tree in which common parts of the paths are stored, without duplication, in the upper parts of the tree. We will use the following representation of such a tree in the program. We have two cases:

Case 1: The tree only consists of a single node, N; then it is represented by the term **l(N)**; the functor **l** is intended to indicate that N is a leaf node in the tree.

Case 2: The tree consists of a root node, N, and a set of its subtrees S1, S2, ... Such a tree will be represented by a term

t(N, Subs)

where Subs is a list of subtrees:

Subs = [S1, S2, ...]

As an example, let us consider the situation when three levels of the tree in Figure 11.9 have been generated. In the list representation, the set of candidate paths is at this point:

[[d,b,a], [e,b,a], [f,c,a], [g,c,a]]

In the tree representation this set of candidate paths is represented by:

t(a, [t(b,[l(d),l(e)]), t(c,[l(f),l(g)])])

This may appear even more wasteful than the list representation, but this is only the surface appearance due to the compact Prolog notation for lists.

In the list representation of the candidate set, the breadth-first effect was achieved by moving the last expanded paths to the end of the candidate list. We cannot use the same trick with the tree representation, therefore the program will be somewhat more complicated. The key relation here will be:

expand(Path, Tree, Tree1, Solved, Solution)

Figure 11.12 illustrates the relation between the arguments of **expand**. Whenever the **expand** procedure is called, **Path** and **Tree** will be already instantiated. **Tree** is a subtree of the search tree and represents the set of candidate paths to a goal in this subtree. **Path** is the path between the start node and the root of

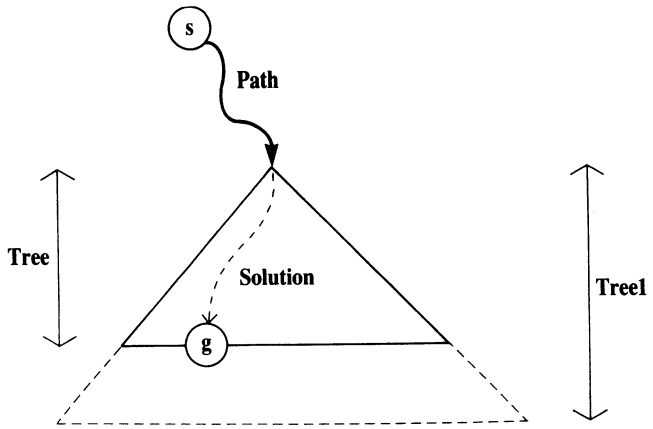


Figure 11.12 Relation **expand(Path, Tree, Tree1, Solved, Solution)**: **s** is the start node of the search and **g** is a goal node. **Solution** is **Path** extended to **g**. **Tree1** is **Tree** grown by one level downwards.

Tree. The general idea of expand is to produce **Tree1** as a one-level extension of **Tree**. If, however, during the expansion of **Tree** a goal node is encountered, **expand** will produce the corresponding solution path.

Thus, **expand** will produce two kinds of results. The kind of result will be indicated by the value of **Solved** as follows:

(1) **Solved** = **yes**.
Solution = solution path; that is, **Path** extended to a goal.
Tree1 = uninstantiated.

Of course, results of this kind are only produced when there is a goal node in **Tree**. In addition, such a goal node has to be a leaf in **Tree**.

(2) **Solved** = **no**.
Tree1 = **Tree** expanded by one level at its bottom edge. **Tree1** does not contain any 'dead' branches in **Tree**; that is, branches that cannot be further extended because they have no successor or a cycle would be created.
Solution = uninstantiated.

If there is no goal node in **Tree**, and **Tree** cannot be expanded, then **expand** fails.

The top-level, breadth-first procedure

breadthfirst(Tree, Solution)

finds **Solution** in the candidate set **Tree** or in its extension. Figure 11.13 shows the complete program. An auxiliary procedure in this program is **expandall** which is similar to **expand**. It expands a *list* of trees and collects the resulting

```
% BREADTH-FIRST SEARCH
% Candidate set is represented as a tree

solve( Start, Solution) :-
  breadthfirst( l(Start), Solution).

breadthfirst( Tree, Solution) :-
  expand( [], Tree, Tree1, Solved, Solution),
  ( Solved = yes;
    Solved = no, breadthfirst( Tree1, Solution) ).

expand( P, l(N), _, yes, [N | P] ) :-
  goal( N).

expand( P, l(N), t( N, Subs), no, _) :-
  bagof( l(M), ( s(N,M), not member(M,P) ), Subs).

expand( P, t( N, Subs), t( N, Subs1), Solved, Sol) :-
  expandall( [N | P], Subs, [], Subs1, Solved, Sol).

expandall( _, [], [T | Ts], [T | Ts], no, _).   % At least one tree must have grown

expandall( P, [T | Ts], Ts1, Subs1, Solved, Sol) :-
  expand( P, T, T1, Solved1, Sol),
  ( Solved1 = yes, Solved = yes;
    Solved1 = no, !,
    expandall( P, Ts, [T1 | Ts1], Subs1, Solved, Sol) );
  expandall( P, Ts, Ts1, Subs1, Solved, Sol).
```

Figure 11.13 An implementation of breadth-first search using the tree representation of the set of candidate paths.

expanded trees, throwing away all 'dead trees'. It also produces, by backtracking, all solutions found in this list of trees. There is an additional detail: at least one of the trees must have grown. If not, then **expandall** will not succeed in producing any expanded trees because all trees in the list are 'dead'.

We have developed this more sophisticated implementation of breadth-first search not only because it is more economical than our previous version, but also because this solution is a good start for developing more sophisticated, heuristically guided search programs, such as best-first search of Chapter 12.

Exercises

11.5 Rewrite the breadth-first program of Figure 11.10 using the difference-pair representation for the list of candidate paths, and show that the

result can be the program in Figure 11.11. In Figure 11.11, what is the purpose of the goal:

Paths \== Z

Test what happens if this goal is omitted on the state space of Figure 11.9. The difference should only show when trying to find more solutions when there are none left.

11.6 How can the search programs of this section be used for searching from a *starting set* of nodes instead of a single start node?

11.7 How can the search programs of this chapter be used to search in the backward direction; that is, starting from a goal node and progressing toward the start node (or *a* start node in the case of multiple start nodes). Hint: Redefine the **s** relation. In what situations would the backward search be advantageous over the forward search?

11.8 Sometimes it is beneficial to search *bidirectionally*; that is, to work from both ends, the start and the goal. The search ends when both ends come together. Define the search space (relation **s**) and the goal relation for a given graph so that our search procedures would in effect perform bidirectional search.

11.9 Experiment with various search techniques in the blocks world planning problem.

11.4 Comments on searching graphs, on optimality, and on search complexity

At this point it is appropriate to make some comments on our search programs so far: first, on searching graphs; second, on the optimality of solutions produced; and third, on the complexity of search.

Examples so far might have made the wrong impression that our breadth-first programs only work for state spaces that are trees and not general graphs. The fact that the set of candidate paths was in one version represented as a tree does not mean that the state space itself had to be a tree. When a graph is searched it, in effect, unfolds into a tree so that some paths are possibly copied in other parts of the tree. Figure 11.14 illustrates.

Our breadth-first search programs generate solution paths, one after another, ordered according to their lengths: shortest solutions come first. This is important if optimality (with respect to length) is a concern. The breadth-first strategy is guaranteed to produce a shortest solution first. This is, of course, not true for the depth-first strategy.

Our programs do not, however, take into account any costs associated

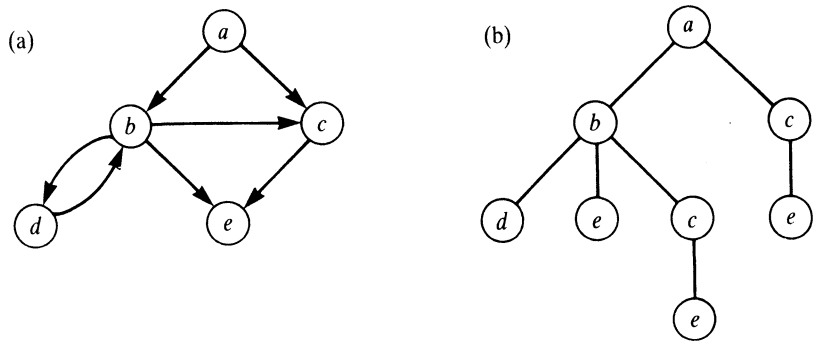


Figure 11.14 (a) A state space: *a* is the start node. (b) The tree of all possible non-cyclic paths from *a*, as developed by the breadth-first search program of Figure 11.13.

with the arcs in the state space. If the minimal cost of the solution path is the optimization criterion (and not its length) then the breadth-first search is not sufficient. The best-first search of Chapter 12 will aspire to optimize the cost.

Another typical problem associated with searching problem spaces is that of the *combinatorial complexity*. For non-trivial problem domains the number of alternatives to be explored is so high that the problem of complexity often becomes critical. It is easy to understand why this happens: If each node in the state space has b successors then the number of paths of length l from the start node is b^l (we assume no cycles). Thus the set of candidate solution paths grows *exponentially* with their length, which leads to what is called the *combinatorial explosion*. The depth-first and breadth-first strategies do not do anything clever against this complexity: they non-selectively treat all candidates as equally promising.

A more sophisticated search procedure should use some problem-specific information to decide what is the most promising way to proceed at each stage of the search. This should have the effect of drawing the search process toward a goal node, avoiding aimless paths. Problem-specific information that can be thus used for guiding the search is called *heuristic*. Algorithms that use heuristics are said to be *heuristically guided*: they perform *heuristic search*. The next chapter presents such a search method.

Summary

- *State space* is a formalism for representing problems.
- State space is a directed graph whose nodes correspond to problem situations and arcs to possible moves. A particular problem is defined by a *start node* and a *goal condition*. A solution of the problem then corresponds to a path in the graph. Thus problem solving is reduced to searching for a path in a graph.

- Optimization problems can be modelled by attaching costs to the arcs of a state space.
- Two basic search strategies that systematically explore a state space are *depth first* and *breadth first*.
- The depth-first search is easiest to program, but is susceptible to cycling. Two simple methods to prevent cycling are: limit the depth of search; test for repeated nodes.
- Implementation of the breadth-first strategy is more complicated as it requires maintaining the set of candidates. This can be most easily represented as a list of lists, but more economically as a tree.
- The breadth-first search always finds a shortest solution path first, but this is not the case with the depth-first strategy.
- In the case of large state spaces there is the danger of *combinatorial explosion*. Both depth-first and breadth-first strategies are poor tools in combatting this difficulty. Heuristic guidance is required in such cases.
- Concepts introduced in this chapter are:

  state space
  start node, goal condition, solution path
  search strategy
  depth-first search, breadth-first search
  heuristic search

References

Depth-first and breadth-first search are basic search strategies described in any general text on Artificial Intelligence; see, for example, Nilsson (1971, 1980) or Winston (1984). Kowalski (1980) showed how logic can be used for implementing these principles.

Kowalski, R. (1980) *Logic for Problem Solving*. Prentice-Hall.

Nilsson, N. J. (1971) *Problem Solving Methods in Artificial Intelligence*. McGraw-Hill.

Nilsson, N. J. (1980) *Principles of Artificial Intelligence*. Tioga; also Springer-Verlag, 1981.

Winston, P. H. (1984) *Artificial Intelligence* (second edition). Addison-Wesley.

12 Best First: A Heuristic Search Principle

Graph searching in problem solving typically leads to the problem of combinatorial complexity due to the proliferation of alternatives. Heuristic search aspires to fight this problem efficiently.

One way of using heuristic information about a problem is to compute numerical *heuristic estimates* for the nodes in the state space. Such an estimate of a node indicates how promising a node is with respect to reaching a goal node. The idea is to continue the search always from the most promising node in the candidate set. The best-first search program of this chapter is based on this principle.

12.1 Best-first search

A best-first search program can be derived as a refinement of the breadth-first search program of Figure 11.13. The best-first search also starts at the start node and maintains the set of candidate paths. The breadth-first search always chooses for expansion a shortest candidate path (that is, shallowest tip nodes of the search). The best-first search refines this principle by computing a heuristic estimate for each candidate and chooses for expansion the best candidate according to this estimate.

We will from now on assume that a cost function is defined for the arcs of the state space. So $c(n,n')$ is the cost of moving from a node n to its successor n' in the state space.

Let the heuristic estimator be a function f, such that for each node n of the space, $f(n)$ estimates the 'difficulty' of n. Accordingly, the most promising current candidate node is the one that minimizes f. We will use here a specially constructed function f which leads to the well-known A* algorithm. $f(n)$ will be constructed so as to estimate the cost of a best solution path from the start node, s, to a goal node, under the constraint that this path goes through n. Let us suppose that there is such a path and that a goal node that minimizes its cost is t. Then the estimate $f(n)$ can be constructed as the sum of two terms, as illustrated in Figure 12.1:

$$f(n) = g(n) + h(n)$$

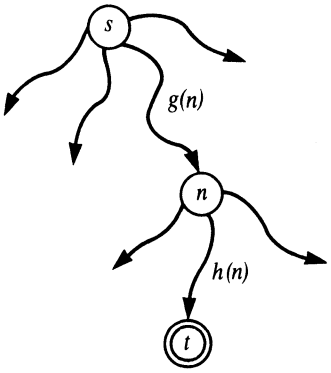


Figure 12.1 Construction of a heuristic estimate $f(n)$ of the cost of the cheapest path from s to t via n: $f(n) = g(n) + h(n)$.

$g(n)$ is an estimate of the cost of an optimal path from s to n; $h(n)$ is an estimate of the cost of an optimal path from n to t.

When a node n is encountered by the search process we have the following situation: a path from s to n must have already been found, and its cost can be computed as the sum of the arc costs on the path. This path is not necessarily an optimal path from s to n (there may be a better path from s to n not, yet, found by the search), but its cost can serve as an estimate $g(n)$ of the minimal cost from s to n. The other term, $h(n)$, is more problematic because the 'world' between n and t has not been explored by the search until this point. Therefore, $h(n)$ is typically a real heuristic guess, based on the algorithm's general knowledge about the particular problem. As h depends on the problem domain there is no universal method for constructing h. Concrete examples of how such a heuristic guess can be made will be shown later. But let us assume for now that a function h is given, and concentrate on details of our best-first program.

We can imagine the best-first search to work as follows. The search process consists of a number of competing subprocesses, each of them exploring its own alternative; that is, exploring its own subtree. Subtrees have subtrees: these are explored by subprocesses of subprocesses, etc. Among all these competing processes, only one is active at each time: the one that deals with the currently most promising alternative; that is, the alternative with the lowest f-value. The remaining processes have to wait quietly until the current f-estimates change so that some other alternative becomes more promising. Then the activity is switched to this alternative. We can imagine this activate–deactivate mechanism as functioning as follows: the process working on the currently top-priority alternative is given some budget and the process is active until this budget is exhausted. During this activity, the process keeps expanding its subtree and reports a solution if a goal node was encountered. The budget for this run is defined by the heuristic estimate of the closest competing alternative.

Figure 12.2 shows an example of such behaviour. Given a map, the task is

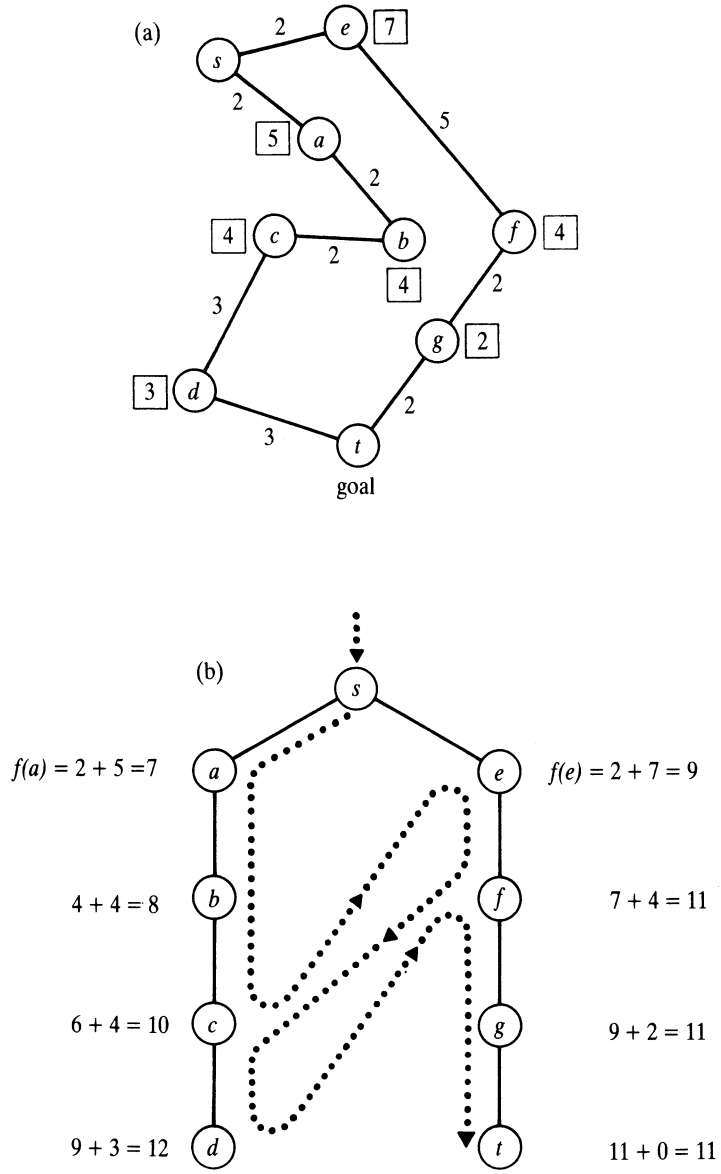


Figure 12.2 Finding the shortest route from s to t in a map. (a) The map with links labelled by their lengths; the numbers in the boxes are straight-line distances to t. (b) The order in which the map is explored by a best-first search. Heuristic estimates are based on straight-line distances. The dotted line indicates the switching of activity between alternative paths. The line shows the order in which the nodes are *expanded*, not the order in which they are generated.

to find the shortest route between the start city s and the goal city t. In estimating the cost of the remaining route distance from a city X to the goal we simply use the straight-line distance denoted by $dist(X,t)$. So:

$$f(X) = g(X) + h(X) = g(X) + dist(X,t)$$

In this example, we can imagine the best-first search as consisting of two processes, each of them exploring one of the two alternative paths: Process 1 the path via a, Process 2 the path via e. In initial stages, Process 1 is more active because f-values along its path are lower than along the other path. At the moment that Process 1 is at c and Process 2 still at e, the situation changes:

$$f(c) = g(c) + h(c) = 6 + 4 = 10$$
$$f(e) = g(e) + h(e) = 2 + 7 = 9$$

So $f(e) < f(c)$, and now Process 2 proceeds to f and Process 1 waits. Here, however,

$$f(f) = 7 + 4 = 11$$
$$f(c) = 10$$
$$f(c) < f(f)$$

Therefore Process 2 is stopped and Process 1 allowed to proceed, but only to d when $f(d) = 12 > 11$. Process 2, invoked at this point, now runs smoothly up to the goal t.

We will program this behaviour as a refinement of the breadth-first search program of Figure 11.13. The set of candidate paths will be, again, represented as a tree. This tree will be represented in the program by terms of two forms:

(1) **l(N, F/G)** represents a single node tree (a leaf); N is a node in the state space, G is g(N) (cost of the path found from the start node to N); F is f(N) = G + h(N).

(2) **t(N, F/G, Subs)** represents a tree with non-empty subtrees; N is the root of the tree, **Subs** is a list of its subtrees; G is g(N); F is the *updated* f-value of N – that is, the f-value of the most promising successor of N; the list **Subs** is ordered according to increasing f-values of the subtrees.

The updating of the f-values is necessary to enable the program to recognize the most promising subtree at each level of the search tree (that is, the tree that contains the most promising tip node). This modification of f-estimates leads, in fact, to a generalization of the definition of f. The generalization extends the definition of the function f from nodes to trees. For a single node tree (a leaf), n, we have the original definition:

$$f(n) = g(n) + h(n)$$

For a tree, T, whose root is n, and n's successors are m_1, m_2, etc.,

$$f(T) = \min_i f(m_i)$$

A best-first program along these lines is shown as Figure 12.3. Some more explanation of this program follows.

```
% Best-first search

bestfirst( Start, Solution) :-
  biggest( Big),                              % Big > any f-value
  expand( [], l( Start, 0/0), Big, _, yes, Solution).

expand( P, l( N, _), _, _, yes, [N | P] ) :-
  goal( N).

expand( P, l( N, F/G), Bound, Tree1, Solved, Sol) :-
  F =< Bound,
  ( bagof( M/C, ( s(N,M,C), not member(M,P) ), Succ), !,
       succlist( G, Succ, Ts),
       bestf( Ts, F1),
       expand( P, t( N, F1/G,Ts), Bound, Tree1, Solved, Sol);
    Solved = never).                          % No successors - dead end

expand( P, t( N, F/G, [T | Ts] ), Bound, Tree1, Solved, Sol) :-
  F =< Bound,
  bestf( Ts, BF), min( Bound, BF, Bound1),
  expand( [N | P], T, Bound1, T1, Solved1, Sol),
  continue( P, t( N, F/G, [T1 | Ts] ), Bound, Tree1, Solved1, Solved, Sol).

expand( _, t( _, _, [] ), _, _, never, _) :- !.  % A dead tree will never be solved

expand( _, Tree, Bound, Tree, no, _) :-
  f( Tree, F), F > Bound.                     % Cannot grow - bound exceeded

continue( _, _, _, _, yes, yes, Sol).

continue( P, t( N, F/G, [T1 | Ts] ), Bound, Tree1, Solved1, Solved, Sol) :-
  ( Solved1 = no, insert( T1, Ts, NTs);
    Solved1 = never, NTs = Ts),
  bestf( NTs, F1),
  expand( P, t( N, F1/G, NTs), Bound, Tree1, Solved, Sol).

succlist( _, [], [] ).

succlist( G0, [N/C | NCs], Ts) :-
  G is G0 + C,
  h( N, H),                                   % Heuristic term h(N)
  F is G + H,
  succlist( G0, NCs, Ts1),
  insert( l( N, F/G), Ts1, Ts).
```

```
% Insert T into list of trees Ts preserving order w.r.t. f-values

insert( T, Ts, [T | Ts] ) :-
  f( T, F), bestf( Ts, F1),
  F =< F1, !.

insert( T, [ T1 | Ts], [ T1 | Ts1] ) :-
  insert( T, Ts, Ts1).

% Extract f-value

f( l( _, F/_ ), F).                    % f-value of a leaf

f( t( _, F/_, _ ), F).                 % f-value of a tree

bestf( [ T | _ ], F) :-                % Best f-value of a list of trees
  f( T, F).

bestf( [], Big) :-                     % No trees: bad f-value
  biggest( Big).

min( X, Y, X) :-
  X =< Y, !.

min( X, Y, Y).
```

Figure 12.3 A best-first search program.

As in the breadth-first search of Figure 11.13, the key procedure is **expand**, which has six arguments this time:

expand(P, Tree, Bound, Tree1, Solved, Solution)

It expands a current (sub)tree as long as the *f*-value of this tree remains less or equal to **Bound**. The arguments of **expand** are:

P	Path between the start node and **Tree**.
Tree	Current search (sub)tree.
Bound	*f*-limit for expansion of *Tree*.
Tree1	**Tree** expanded within **Bound**; consequently, the *f*-value of **Tree1** is greater than **Bound** (unless a goal node has been found during the expansion).
Solved	Indicator whose value is 'yes', 'no' or 'never'.
Solution	A solution path from the start node 'through **Tree1**' to a goal node within **Bound** (if such a goal node exists).

P, **Tree** and **Bound** are 'input' parameters to **expand**; that is, they are already

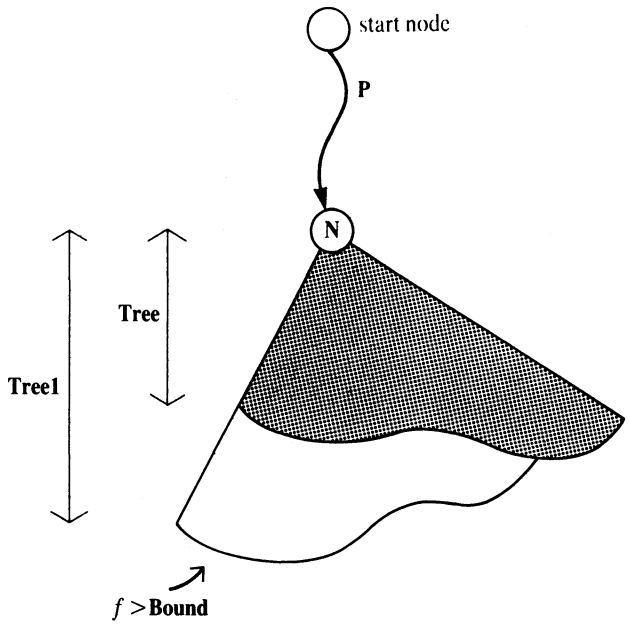


Figure 12.4 The **expand** relation: expanding **Tree** until the *f*-value exceeds **Bound** results in **Tree1**.

instantiated whenever **expand** is called. **expand** produces three kinds of results, which is indicated by the value of the argument **Solved** as follows:

(1) **Solved** = **yes**.
Solution = a solution path found by expanding **Tree** within **Bound.**
Tree1 = uninstantiated.

(2) **Solved** = **no**.
Tree1 = **Tree** expanded so that its *f*-value exceeds **Bound** (Figure 12.4 illustrates).
Solution = uninstantiated.

(3) **Solved** = **never**.
Tree1 and **Solution** = uninstantiated.

The last case indicates that **Tree** is a 'dead' alternative and should never be given another chance by reactivating its exploration. This case arises when the *f*-value of **Tree** is less or equal to **Bound**, but the tree cannot grow because no leaf in it has any successor at all, or such a successor would create a cycle.

Some clauses about **expand** deserve explanation. The clause that deals with the most complicated case when **Tree** has subtrees – that is,

Tree = **t(N, F/G, [T | Ts])**

says the following. First, the most promising subtree, T, is expanded. This expansion is not given the bound **Bound**, but possibly some lower value, depending on the *f*-values of the other competing subtrees, **Ts**. This ensures that the currently growing subtree is always the most promising subtree. The expansion process then switches between the subtrees according to their *f*-values. After the best candidate has been expanded, an auxiliary procedure **continue** decides what to do next; this depends on the type of result produced by this expansion. If a solution was found then this is returned, otherwise expansion continues.

The clause that deals with the case

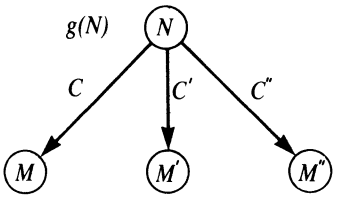
Tree = l(N, F/G)

generates successor nodes of N together with the costs of the arcs between N and successor nodes. Procedure **succlist** makes a list of subtrees from these successor nodes, also computing their *g*-values and *f*-values as shown in Figure 12.5. The resulting tree is then further expanded as far as **Bound** permits. If, on the other hand, there were no successors, then this leaf is abandoned for ever by instantiating **Solved** = 'never'.

Other relations are:

s(N, M, C)	M is a successor node of N in the state space; C is the cost of the arc from N to M.
h(N, H)	H is a heuristic estimate of the cost of the best path from node N to a goal node.
biggest(Big)	**Big** is some user-specified value that is known to be greater than any possible *f*-value.

Application of this best-first search program to some example problems will be shown in the next section. But first some general, concluding comments on this program. It is a variation of a heuristic algorithm known in the literature as the A* algorithm (see references at the end of the chapter). A* has attracted



$g(M) = g(N) + C$
$f(M) = g(M) + h(M)$

Figure 12.5 Relation between the *g*-value of node *N*, and the *f*- and *g*-values of its children in the search space.

a great deal of attention; we will mention here an important result from the mathematical analysis of A*:

A search algorithm is said to be *admissible* if it always produces an optimal solution (that is, a minimum-cost path) provided that a solution exists at all. Our implementation, which produces all solutions through backtracking, can be considered admissible if the *first* solution found is optimal. Let, for each node n in the state space, $h^*(n)$ denote the cost of an optimal path from n to a goal node. A theorem about the admissibility of A* says: an A* algorithm that uses a heuristic function h such that for all nodes n in the state space

$$h(n) \leqq h^*(n)$$

is admissible.

This result is of great practical value. Even if we do not know the exact value of h^* we just have to find a lower bound of h^* and use it as h in A*. This is sufficient guarantee that A* will produce an optimal solution.

There is a trivial lower bound, namely:

$$h(n) = 0, \quad \text{for all } n \text{ in the state space}$$

This indeed guarantees admissibility. The disadvantage of $h = 0$ is, however, that it has no heuristic power and does not provide any guidance for the search. A* using $h = 0$ behaves similarly to the breadth-first search. It in fact reduces to the breadth-first search in the case that the arc-cost function $c(n,n') = 1$ for all arcs (n,n') in the state space. The lack of heuristic power results in high complexity. We would therefore like to have h, which is a lower bound of h^* (to ensure admissibility), and which is also as close as possible to h^* (to ensure efficiency). Ideally, if we knew h^*, we would use h^* itself: A* using h^* finds an optimal solution directly, without any backtracking at all.

Exercise

12.1 Define the problem-specific relations **s**, **goal** and **h** for the route-finding problem of Figure 12.2. Inspect the behaviour of our best-first search program on this problem.

12.2 Best-first search applied to the eight puzzle

If we want to apply the best-first search program of Figure 12.3 to some particular problem we have to add problem-specific relations. These relations define the particular problem ('rules of the game') and also convey heuristic

information about how to solve that problem. This heuristic information is supplied in the form of a heuristic function.

Problem-specific predicates are:

s(Node, Node1, Cost)

This is true if there is an arc, costing **Cost**, between **Node** and **Node1** in the state space.

goal(Node)

is true if **Node** is a goal node in the state space.

h(Node, H)

H is a heuristic estimate of the cost of a cheapest path from **Node** to a goal node.

In this and the following sections we will define these relations for two example problem domains: the eight puzzle (described in Section 11.1) and the task-scheduling problem.

Problem-specific relations for the eight puzzle are shown in Figure 12.6. A node in the state space is some configuration of the tiles on the board. In the program, this is represented by a list of the current positions of the tiles. Each position is specified by a pair of coordinates: **X/Y**. The order of items in the list is as follows:

(1) the current position of the empty square,
(2) the current position of tile 1,
(3) the current position of tile 2,
...

The goal situation (see Figure 11.3) is defined by the clause:

goal([2/2,1/3,2/3,3/3,3/2,3/1,2/1,1/1,1/2]).

An auxiliary relation is:

d(S1, S2, D)

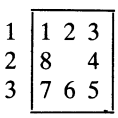
D is the 'Manhattan distance' between squares S1 and S2; that is, the distance between S1 and S2 in the horizontal direction plus the distance between S1 and S2 in the vertical direction.

We want to minimize the *length* of solutions. Therefore, we define the cost of all the arcs in the state space to equal 1. In the program of Figure 12.6, three example starting positions from Figure 12.7 are also defined.

/* Problem-specific procedures for the eight puzzle

Current situation is represented as a list of positions of the tiles, with first item in the list corresponding to the empty square.

Example:



This position is represented by:

[2/2, 1/3, 2/3, 3/3, 3/2, 3/1, 2/1, 1/1]

'Empty' can move to any of its neighbours which means that 'empty' and its neighbour interchange their positions.
*/

```
s( [Empty | L], [T | L1], 1) :-          % All arc costs are 1
  swap( Empty, T, L, L1).                % Swap Empty and T in L giving L1

swap( E, T, [T | L], [E | L] ) :-
  d( E, T, 1).

swap( E, T, [T1 | L], [T1 | L1] ) :-
  swap( E, T, L, L1).

d( X/Y, X1/Y1, D) :-                     % D is Manh. dist. between two squares
  dif( X, X1, Dx),
  dif( Y, Y1, Dy),
  D is Dx + Dy.

dif( A, B, D) :-
  D is A-B, D >= 0, !;
  D is B-A.

% Heuristic estimate h is the sum of distances of each tile
% from its 'home' square plus 3 times 'sequence' score

h( [Empty | L], H) :-
  goal( [Empty1 | G] ),,
  totdist( L, G, D),
  seq( L, S),
  H is D + 3*S.

totdist( [], [], 0).

totdist( [T | L], [T1 | L1], D) :-
  d( T, T1, D1),
  totdist( L, L1, D2),
  D is D1 + D2.

seq( [First | L], S) :-
  seq( [First | L ], First, S).
```

```
seq( [T1, T2 | L], First, S) :-
  score( T1, T2, S1),
  seq( [T2 | L], First, S2),
  S is S1 + S2.

seq( [Last], First, S) :-
  score( Last, First, S).

score( 2/2, _, 1) :- !.                          % Tile in centre scores 1

score( 1/3, 2/3, 0) :- !.                        % Proper successor scores 0
score( 2/3, 3/3, 0) :- !.
score( 3/3, 3/2, 0) :- !.
score( 3/2, 3/1, 0) :- !.
score( 3/1, 2/1, 0) :- !.
score( 2/1, 1/1, 0) :- !.
score( 1/1, 1/2, 0) :- !.
score( 1/2, 1/3, 0) :- !.

score( _, _, 2).                                 % Tiles out of sequence

goal( [2/2,1/3,2/3,3/3,3/2,3/1,2/1,1/1,1/2] ).

% Starting positions for some puzzles
start1( [2/2,1/3,3/2,2/3,3/3,3/1,2/1,1/1,1/2] ).   % Requires 4 steps
start2( [2/1,1/2,1/3,3/3,3/2,3/1,2/2,1/1,2/3] ).   % 5 steps
start3( [2/2,2/3,1/3,3/1,1/2,2/1,3/3,1/1,3/2] ).   % 18 steps

% Display a solution path as a list of board positions
showsol( [] ).

showsol( [P | L] ) :-
  showsol( L),
  nl, write( '---'),
  showpos( P).

% Display a board position
showpos( [S0,S1,S2,S3,S4,S5,S6,S7,S8] ) :-
  member( Y, [3,2,1] ),                          % Order of Y-coordinates
  nl, member( X, [1,2,3] ),                      % Order of X-coordinates
  member( Tile-X/Y, [' '-S0,1-S1,2-S2,3-S3,4-S4,5-S5,6-S6,7-S7,8-S8] ),
  write( Tile),
  fail.                                          % Backtrack to next square

showpos( _).
```

Figure 12.6 Problem-specific procedures for the eight puzzle, to be used in best-first search of Figure 12.3.

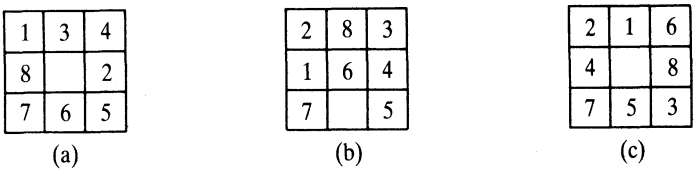


Figure 12.7 Three starting positions for the eight puzzle: (a) requires four steps; (b) requires five steps; (c) requires 18 steps.

The heuristic function, h, is programmed as:

h(Pos, H)

Pos is a board position; H is a combination of two measures:

(1) **totdist**: the 'total distance' of the eight tiles in **Pos** from their 'home squares'. For example, in the starting position of the puzzle in Figure 12.7(a), **totdist** = 4.

(2) **seq**: the 'sequence score' that measures the degree to which the tiles are already ordered in the current position with respect to the order required in the goal configuration. **seq** is computed as the sum of scores for each tile according to the following rules:

- a tile in the centre scores 1;
- a tile on a non-central square scores 0 if the tile is, in the clockwise direction, followed by its proper successor;
- such a tile scores 2 if it is not followed by its proper successor.

For example, for the starting position of the puzzle in Figure 12.7(a), **seq** = 6.

The heuristic estimate, H, is computed as:

H = totdist + 3 * seq

This heuristic function works well in the sense that it very efficiently directs the search toward the goal. For example, when solving the puzzles of Figure 12.7(a) and (b), no node outside the shortest solution path is ever expanded before the first solution is found. This means that the shortest solutions are found directly in these cases without any backtracking. Even the difficult puzzle of Figure 12.7(c) is solved almost directly. A drawback of this heuristic is, however, that it is not admissible: it does not guarantee that the shortest solution path will always be found before any longer solution. The h function used does not satisfy the admissibility condition: $h \leqq h^*$ for all the nodes. For example, for the initial position in Figure 12.7(a),

$h = 4 + 3 * 6 = 22, \quad h^* = 4$

On the other hand, the 'total distance' measure itself is admissible: for all positions

totdist $\leqq h^*$

This relation can be easily proved by the following argument: if we relaxed the problem by allowing the tiles to climb on top of each other, then each tile could travel to its home square along a trajectory whose length is exactly the Manhattan distance between the tile's initial square and its home square. So the optimal solution in the relaxed puzzle would be exactly of length **totdist**. In the original problem, however, there is interaction between the tiles and they are in each other's way. This can prevent the tiles from moving along the shortest trajectories, which ensues our optimal solution's length be equal or greater than **totdist**.

Exercise

12.2 Modify the best-first search program of Figure 12.3 to count the number of nodes generated in the search. One easy way is to keep the current number of nodes asserted as a fact, and update it by **retract** and **assert** whenever new nodes are generated. Experiment with various heuristic functions for the eight puzzle with respect to their heuristic power, which is reflected in the number of nodes generated.

12.3 Best-first search applied to scheduling

Let us consider the following task-scheduling problem. We are given a collection of *tasks*, t_1, t_2, ..., with their execution times D_1, D_2, ... respectively. The tasks are to be executed on a set of m identical *processors*. Any task can be executed on any processor, but each processor can only execute one task at a time. There is a precedence relation between tasks which tells what tasks, if any, have to be completed before some other task can be started. The scheduling problem is to assign tasks to processors so that the precedence relation is not violated and that all the tasks together are processed in the shortest possible time. The time that the last task in a schedule is completed is called the *finishing time* of the schedule. We want to minimize the finishing time over all permissible schedules.

Figure 12.8 shows such a task-scheduling problem and two permissible schedules, one of which is optimal. This example shows an interesting property of optimal schedules; namely, that they may include 'idle time' for processors. In the optimal schedule of Figure 12.8, processor 1 after having executed task t_2 waits for two time units although it could start executing task t_7.

One way to construct a schedule is roughly as follows. We start with the

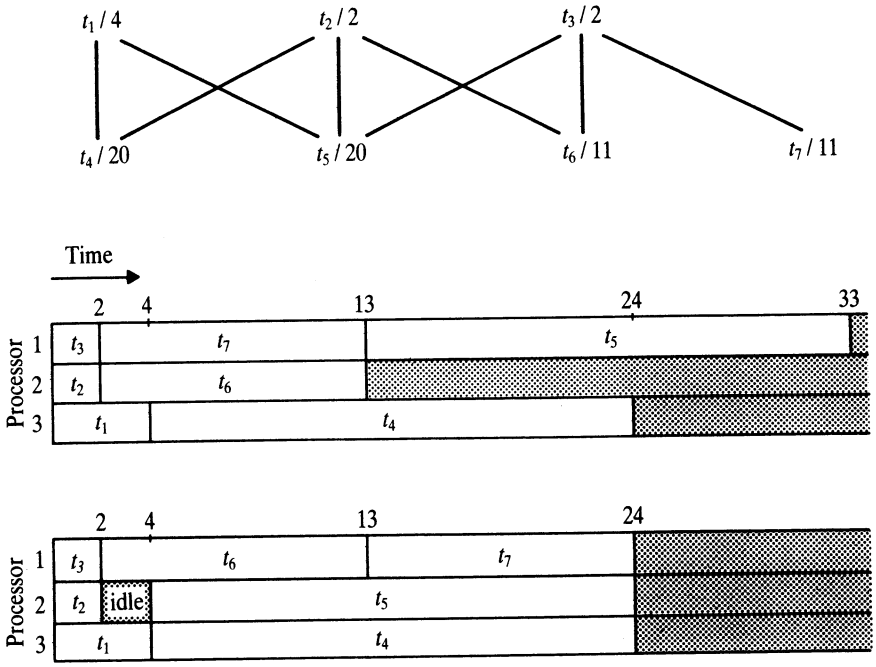


Figure 12.8 A task-scheduling problem with seven tasks and three processors. The top part of the diagram shows the task precedence relation and the duration of the tasks. Task t_5, for example, requires 20 time units, and its execution can only start after three other tasks, t_1, t_2 and t_3, have been completed. Two permissible schedules are shown; an optimal one with the finishing time 24, and a suboptimal one with the finishing time 33. In this problem any optimal schedule has to include idle time. Coffman/Denning, *Operating Systems Theory*, © 1973, p. 86. Adapted by permission of Prentice-Hall, Englewood Cliffs, New Jersey.

empty schedule (with void time slots for each processor) and gradually insert tasks one by one into the schedule until all the tasks have been inserted. Usually there are alternatives at any such insertion step because there are several candidate tasks waiting to be processed. Therefore, the scheduling problem is one of search. Accordingly, we can formulate the scheduling problem as a state-space search problem as follows:

- states are partial schedules;
- a successor state of some partial schedule is obtained by adding a not yet scheduled task to this schedule; another possibility is to leave a processor that has completed its current task idle;
- the start state is the empty schedule;
- any schedule that includes all the tasks in the problem is a goal state;
- the cost of a solution (which is to be minimized) is the finishing time of a goal schedule;

- accordingly, the cost of a transition between two (partial) schedules whose finishing times are F_1 and F_2 respectively is the difference $F_2 - F_1$.

Some refinements are needed to this rough scenario. First, we decide to fill the schedule according to increasing times so that tasks are inserted into the schedule from left to right. Also, each time a task is added, the precedence constraint has to be checked. Further, there is no point in leaving a processor idle indefinitely if there are still some candidate tasks waiting. So we decide to leave a processor idle only until some other processor finishes its current task, and then consider again assigning a task to it.

Now let us decide on the representation of problem situations – that is, partial schedules. We need the following information:

(1) list of waiting tasks and their execution times,
(2) current engagements of the processors;

We will also add for convenience:

(3) the finishing time of the (partial) schedule; that is, the latest end-time of the current engagements of the processors.

The list of waiting tasks and their execution times will be represented in the program as a list of the form:

[Task1/D1, Task2/D2, ...]

The current engagements of the processors will be represented by a list of tasks currently being processed; that is, pairs of the form:

Task/FinishingTime

There are m such pairs in the list, one for each processor. We will always add a new task to a schedule at the moment that the first current execution is completed. To this end, the list of current engagements will be kept ordered according to increasing finishing times. The three components of a partial schedule (waiting tasks, current engagements and finishing time) will be combined in the program into a single expression of the form:

WaitingList * ActiveTasks * FinishingTime

In addition to this information we have the precedence constraint which will be specified in the program as a relation

prec(TaskX, TaskY)

Now let us consider a heuristic estimate. We will use a rather straightfor-

ward heuristic function, which will not provide a very efficient guidance to the search algorithm. The function will be admissible and will hence guarantee an optimal schedule. It should be noted, however, that a much more powerful heuristic would be needed for large scheduling problems.

Our heuristic function will be an optimistic estimate of the finishing time of a partial schedule completed with all currently waiting tasks. This optimistic estimate will be computed under the assumption that two constraints on the actual schedule be relaxed:

(1) remove the precedence constraint;

(2) allow (unrealistically) that a task can be executed in a distributed fashion on several processors, and that the sum of the execution times of this task over all these processors is equal to the originally specified execution time of this task on a single processor.

Let the execution times of the currently waiting tasks be $D_1, D_2, \ldots$, and the finishing times of the current processors engagements be $F_1, F_2, \ldots$. Such an optimistically estimated finishing time, *Finall*, to complete all the currently active and all the waiting tasks, is:

$$Finall = (\sum_i D_i + \sum_j F_j)/m$$

where m is the number of processors. Let the finishing time of the current partial schedule be:

$$Fin = \max_j(F_j)$$

Then the heuristic estimate H (an extra time needed to complete the partial schedule with the waiting tasks) is:

if $Finall > Fin$ then $H = Finall - Fin$ else $H = 0$

A complete program that defines the state-space relations for task scheduling as outlined above is shown in Figure 12.9. The figure also includes a specification of the particular scheduling problem of Figure 12.8. These definitions can now be used by the best-first search program of Figure 12.3. One of the optimal solutions produced by best-first search in the thus specified problem space is an optimal schedule of Figure 12.8.

Project

In general, scheduling problems are known to be combinatorially difficult. Our simple heuristic function does not provide very powerful guidance. Propose other functions and experiment with them.

```
/* Problem-specific relations for task scheduling
Nodes in the state space are partial schedules specified by:

[ Task1/D1, Task2/D2, ...] * [ Task1/F1, Task2/F2, ...] * FinTime

The first list specifies the waiting tasks and their durations; the second list specifies
the currently executed tasks and their finishing times, ordered so that F1 ≦ F2, F2 ≦
F3 ... .FinTime is the latest completion time of current engagements of the
processors.
*/

s( Tasks1 * [_/F | Active1] * Fin1, Tasks2 * Active2 * Fin2, Cost) :-
  del( Task/D, Tasks1, Tasks2),                    % Pick a waiting task
  not ( member( T/_, Tasks2), before( T, Task) ),  % Check precedence
  not ( member( T1/F1, Active1), F < F1, before( T1, Task) ), %Active tasks too
  Time is F + D,                                   % Finishing time of activated task
  insert( Task/Time, Active1, Active2, Fin1, Fin2),
  Cost is Fin2 – Fin1.

s( Tasks * [_/F | Active1] * Fin, Tasks * Active2 * Fin, 0) :-
  insertidle( F, Active1, Active2). % Leave processor idle

before( T1, T2) :-                  % Task T1 before T2 according to precedence
  prec( T1, T2).

before( T1, T2) :-
  prec( T, T2),
  before( T1, T).

insert( S/A, [T/B | L], [S/A, T/B | L], F, F) :-   % Task lists are ordered
  A =< B, !.

insert( S/A, [T/B | L], [T/B | L1], F1, F2) :-
  insert( S/A, L, L1, F1, F2).

insert( S/A, [], [S/A], _, A).

insertidle( A, [T/B | L], [idle/B, T/B | L] ) :-   % Leave processor idle
  A < B, !.                                        % until first greater
                                                   % finishing time

insertidle( A, [T/B | L], [T/B | L1] ) :-
  insertidle( A, L, L1).

del( A, [A | L], L).                               % Delete item from list

del( A, [B | L], [B | L1] ) :-
  del( A, L, L1).

goal( [] * _ * _).                                 % Goal state: no task waiting
```

```
% Heuristic estimate of a partial schedule is based on an
% optimistic estimate of the final finishing time of this
% partial schedule extended by all the remaining waiting tasks.

h( Tasks * Processors * Fin, H) :-
  totaltime( Tasks, Tottime),          % Total duration of waiting tasks
  sumnum( Processors, Ftime, N),       % Ftime is sum of finishing times
                                       % of processors, N is their number
  Finall is ( Tottime + Ftime)/N,
  ( Finall > Fin, !, H is Finall - Fin; H = 0).

totaltime( [], 0).

totaltime( [_/D | Tasks], T) :-
  totaltime( Tasks, T1),
  T is T1 + D.

sumnum( [], 0, 0).

sumnum( [_/T | Procs], FT, N) :-
  sumnum( Procs, FT1, N1),
  N is N1 + 1,
  FT is FT1 + T.

% A task-precedence graph

prec( t1, t4). prec( t1, t5). prec( t2, t4). prec( t2, t5).

prec( t3, t5). prec( t3, t6). prec( t3, t7).

% A start node

start( [t1/4, t2/2, t3/2, t4/20, t5/20, t6/11, t7/11] * [idle/0, idle/0, idle/0] * 0).
```

Figure 12.9 Problem-specific relations for the task-scheduling problem. The particular scheduling problem of Figure 12.8 is also defined by its precedence graph and an initial (empty) schedule as a start node for search.

Summary

- Heuristic information can be used to estimate how far a node is from a nearest goal node in the state space. This chapter considered numerical heuristic estimates.
- The *best-first* heuristic principle guides the search process so as to always expand the node that is currently the most promising, according to the heuristic estimates.

- The search algorithm, known as A*, that uses this principle, was programmed in this chapter.
- To use A* for solving a concrete problem, a state space and a heuristic function have to be defined to be used by A*. For complex problems, the difficult part is to find a good heuristic function.
- The *admissibility theorem* helps to establish whether A*, using a particular heuristic function, will always find an optimal solution.

References

The best-first search program of this chapter is a variation of many similar algorithms of which A* is the most popular. General descriptions of A* can be found in Nilsson (1971, 1980) or Winston (1984). The admissibility theorem was discovered by Hart, Nilsson and Raphael (1968). An excellent and rigorous treatment of many variations of best-first search algorithms and related mathematical results is provided by Pearl (1984).

The eight puzzle was used in Artificial Intelligence as a test problem for studying heuristic principles by several researchers – for example, Doran and Michie (1966), Michie and Ross (1970), and Gaschnig (1979).

Our task-scheduling problem and its variations arise in numerous applications in which servicing of requests for resources is to be planned. An example is computer operating systems. Task-scheduling problems with reference to this application are treated by Coffman and Denning (1973).

Finding good heuristics is important and difficult, therefore the study of heuristics is one of the central themes of Artificial Intelligence. There are, however, also some limitations on how far we can get in the refinement of heuristics. It may appear that to solve any combinatorial problem efficiently we only have to find a powerful heuristic. However, there are problems (including many scheduling problems) for which no general heuristic exists that would guarantee both efficiency and admissibility in all cases. Many theoretical results that pertain to this limitation issue are collected in Garey and Johnson (1979).

Coffman, E. G. and Denning, P. J. (1973) *Operating Systems Theory*. Prentice-Hall.

Doran, J. and Michie, D. (1966) Experiments with the graph traverser program. *Proc. Royal Society of London* **294(A)**: 235–259.

Garey, M. R. and Johnson, D. S. (1979) *Computers and Intractability*. W. H. Freeman.

Gaschnig, J. (1979) Performance measurement and analysis of certain search algorithms. Carnegie-Mellon University: Computer Science Department. Technical Report CMU-CS-79-124 (Ph.D.Thesis).

Hart, P. E., Nilsson, N. J. and Raphael, B. (1968) A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Sciences and Cybernetics* **SSC-4(2)**: 100–107.

Michie, D. and Ross, R. (1970) Experiments with the adaptive graph traverser. *Machine Intelligence* **5**: 301–308.

Nilsson, N. J. (1971) *Problem Solving Methods in Artificial Intelligence*. McGraw-Hill.

Nilsson, N. J. (1980) *Principles of Artificial Intelligence*. Tioga; also Springer-Verlag.

Pearl, J. (1984) *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley.

Winston, P. H. (1984) *Artificial Intelligence* (second edition). Addison-Wesley.

13 Problem Reduction and AND/OR Graphs

AND/OR graphs are a suitable representation for problems that can be naturally decomposed into mutually independent subproblems. Examples of such problems include route finding, symbolic integration, game playing, theorem proving, etc. In this chapter we will develop programs for searching AND/OR graphs, including a heuristically guided best-first AND/OR search.

13.1 AND/OR graph representation of problems

In Chapters 11 and 12, problem solving was centred around the state-space representation of problems. Accordingly, problem solving was reduced to finding a path in a state-space graph. Another representation, the AND/OR graph representation, more naturally suits certain categories of problems. This representation relies on the decomposition of problems into subproblems.

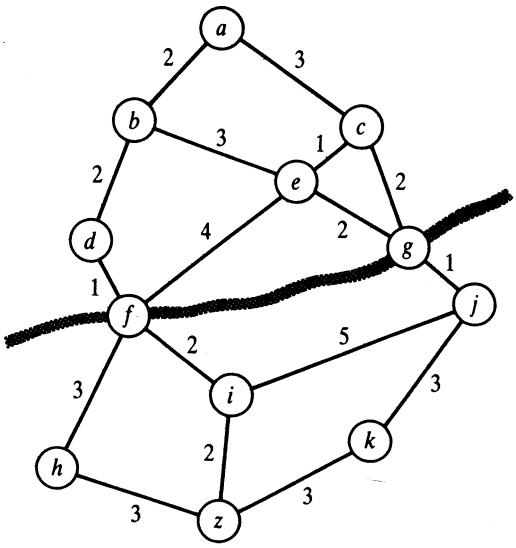


Figure 13.1 Finding a route from a to z in a road map. The river has to be crossed at f or g. An AND/OR representation of this problem is shown in Figure 13.2.

Decomposition into subproblems is advantageous if the subproblems are mutually independent, and can therefore be solved independently of each other.

Let us illustrate this with an example. Consider the problem of finding a route in a road map between two given cities, as shown in Figure 13.1. We will disregard path lengths for the moment. The problem could, of course, be formulated as path finding in a state space. The corresponding state space would look just like the map: the nodes in the state space correspond to cities, the arcs correspond to direct connections between cities, arc costs correspond to distances between cities. However, let us construct another representation of this problem, based on a natural decomposition of the problem.

In the map of Figure 13.1, there is also a river. Let us assume that there are only two bridges at which the river can be crossed, one bridge at city *f* and the other at city *g*. Obviously, our route will have to include one of the bridges; so it will have to go through *f* or through *g*. We have, then, two major alternatives:

To find a path between *a* and *z*, find *either*

(1) a path from *a* to *z* via *f*, *or*

(2) a path from *a* to *z* via *g*.

Each of these two alternative problems can now be decomposed as follows:

(1) To find a path from *a* to *z* via *f*:
  1.1 find a path from *a* to *f*, *and*
  1.2 find a path from *f* to *z*.

(2) To find a path from *a* to *z* via *g*:
  2.1 find a path from *a* to *g*, *and*
  2.2 find a path from *g* to *z*.

To summarize, we have two main alternatives for solving the original problem: (1) via *f* or (2) via *g*. Further, each of these alternatives can be *decomposed* into two subproblems (1.1 and 1.2, or 2.1 and 2.2 respectively). What is important here is that (in both alternatives) each of the subproblems can be solved independently of the other. Such a decomposition can be pictured as an *AND/OR graph* (Figure 13.2). Notice the curved arcs which indicate the AND relationship between subproblems. Of course, the graph in Figure 13.2 is only the top part of a corresponding AND/OR tree. Further decomposition of subproblems could be based on the introduction of additional intermediate cities.

What are goal nodes in such an AND/OR graph? Goal nodes correspond to subproblems that are trivial or 'primitive'. In our example, such a subproblem would be 'find a route from *a* to *c*', for there is a direct connection between cities *a* and *c* in the road map.

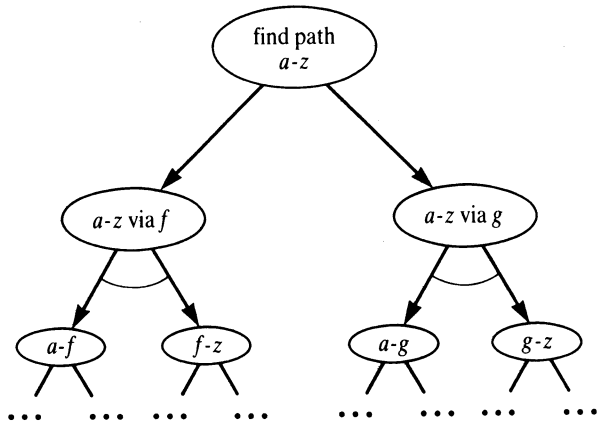


Figure 13.2 An AND/OR representation of the route-finding problem of Figure 13.1. Nodes correspond to problems or subproblems, and curved arcs indicate that all (both) subproblems have to be solved.

Some important concepts have been introduced in this example. An AND/OR graph is a directed graph in which nodes correspond to problems, and arcs indicate relations between problems. There are also relations among arcs themselves. These relations are AND and OR, depending on whether we have to solve just one of the successor problems or several (see Figure 13.3). In principle, a node can issue both AND-related arcs and OR-related arcs. We will, however, assume that each node has either only AND successors or only OR successors. Each AND/OR graph can be transformed into this form by introducing auxiliary OR nodes if necessary. Then, a node that only issues AND arcs is called an AND node; a node that only issues OR arcs is called an OR node.

In the state-space representation, a solution to the problem was a path in the state space. What is a solution in the AND/OR representation? A solution, of course, has to include all the subproblems of an AND node. Therefore, a



Figure 13.3 (a) To Solve P solve any of P_1 or P_2 or... . (b) To solve Q solve all Q_1 and Q_2... .

solution is not a path any more, but it is a tree. Such a solution tree, T, is defined as follows:

- the original problem, P, is the root node of T;
- if P is an OR node then exactly one of its successors (in the AND/OR graph), together with its own solution tree, is in T;
- if P is an AND node then all of its successors (in the AND/OR graph), together with their solution trees, are in T.

Figure 13.4 illustrates this definition. In this figure, there are costs attached to arcs. Using costs we can formulate an optimization criterion. We can, for example, define the cost of a solution graph as the sum of all the arc costs in the graph. As we are normally interested in the minimum cost, the solution graph in Figure 13.4(c) will be preferred.

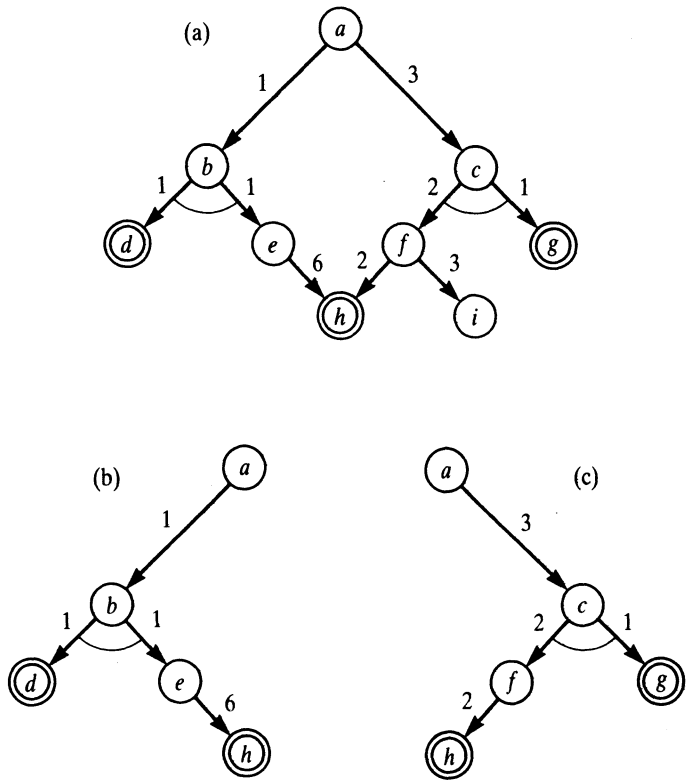


Figure 13.4 (a) An AND/OR graph: *d*, *g* and *h* are goal nodes; *a* is the problem to be solved. (b) and (c) Two solution trees whose costs are 9 and 8 respectively. The cost of a solution tree is here defined as the sum of all the arc costs in the solution tree.

But we do not have to base our optimization measure on the costs of arcs. Sometimes it is more natural to associate costs with nodes rather than arcs, or with both arcs and nodes.

To summarize:

- AND/OR representation is based on the philosophy of reducing a problem to subproblems.
- Nodes in an AND/OR graph correspond to problems; links between nodes indicate relations between problems.
- A node that issues OR links is an OR node. To solve an OR node, one of its successor nodes has to be solved.
- A node that issues AND links is an AND node. To solve an AND node, all of its successors have to be solved.
- For a given AND/OR graph, a particular problem is specified by two things:

  a start node, and
  a goal condition for recognizing goal nodes.
- *Goal nodes* (or 'terminal' nodes) correspond to trivial (or 'primitive') problems.
- A solution is represented by a *solution graph*, a subgraph of the AND/OR graph.
- The state-space representation can be viewed as a special case of the AND/OR representation in which all the nodes are OR nodes.
- To benefit from the AND/OR representation, AND-related nodes should represent subproblems that can be solved independently of each other. The independency criterion can be somewhat relaxed, as follows: there must exist an ordering of AND subproblems so that solutions of subproblems that come earlier in this ordering are not destroyed when solving later subproblems.
- Costs can be attached to arcs or nodes or both in order to formulate an optimization criterion.

13.2 Examples of AND/OR representation

13.2.1 AND/OR representation of route finding

For the shortest route problem of Figure 13.1, an AND/OR graph including a cost function can be defined as follows:

- OR nodes are of the form X-Z, meaning: find a shortest path from X to Z.

- AND nodes are of the form

  X-Z via Y

  meaning: find a shortest path from X to Z under the constraint that the path goes through Y.
- A node X-Z is a goal node (primitive problem) if X and Z are directly connected in the map.
- The cost of each goal node X-Z is the given road distance between X and Z.
- The costs of all other (non-terminal) nodes are 0.

The cost of a solution graph is the sum of the costs of all the nodes in the solution graph (in our case, this is just the sum over the terminal nodes). For the problem of Figure 13.1, the start node is *a-z*. Figure 13.5 shows a solution tree of cost 9. This tree corresponds to the path **[a,b,d,f,i,z]**. This path can be

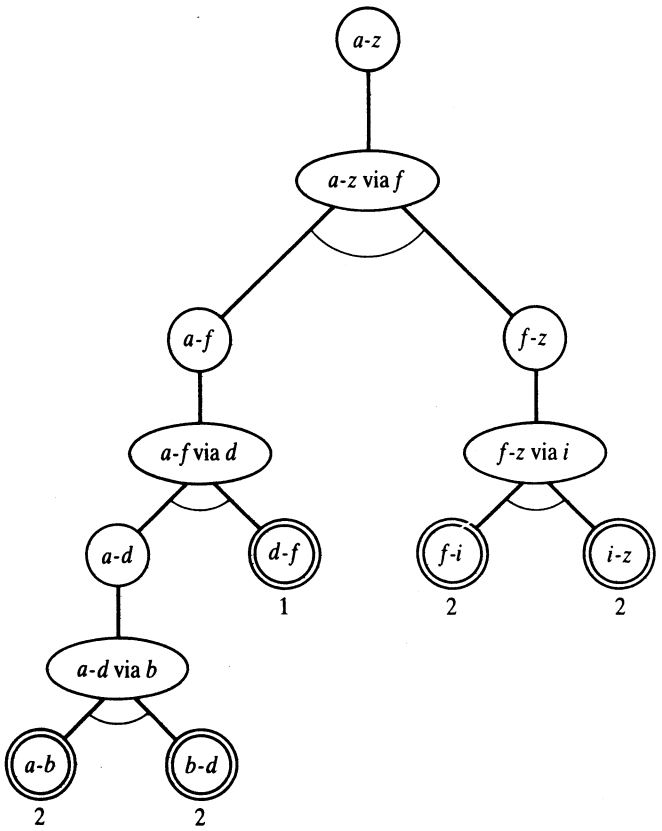


Figure 13.5 The cheapest solution tree for the route problem of Figure 13.1 formulated as an AND/OR graph.

reconstructed from the solution tree by visiting all the leaves in this tree in the left-to-right order.

13.2.2 The Tower of Hanoi problem

The Tower of Hanoi problem, shown in Figure 13.6, is another, classical example of effective application of the AND/OR decomposition scheme. For simplicity, we will consider a simple version of this problem, containing three disks only:

> There are three pegs, 1, 2 and 3, and three disks, *a*, *b* and *c* (*a* being the smallest and *c* being the biggest). Initially, all the disks are stacked on peg 1. The problem is to transfer them all on to peg 3. Only one disk can be moved at a time, and no disk can ever be placed on top of a smaller disk.

This problem can be viewed as the problem of achieving the following set of goals:

(1) Disk *a* on peg 3.
(2) Disk *b* on peg 3.
(3) Disk *c* on peg 3.

These goals are, unfortunately, not independent. For example, disk *a* can immediately be placed on peg 3, satisfying the first goal. This will, however, prevent the fulfilment of the other two goals (unless we undo the first goal again). Fortunately, there is a convenient ordering of these goals so that a solution can easily be derived from this ordering. The ordering can be found by the following reasoning: goal 3 (disk *c* on peg 3) is the hardest because moving disk *c* is subject to most constraints. A good idea that often works in such situations is: try to achieve the hardest goal first. The logic behind this principle is: as other goals are easier (not as constrained as the hardest) they can hopefully be achieved without the necessity of undoing this hardest goal.

The problem-solving strategy that results from this principle in our task is:

> First satisfy the goal 'disk *c* on peg 3',
> then satisfy the remaining goals.

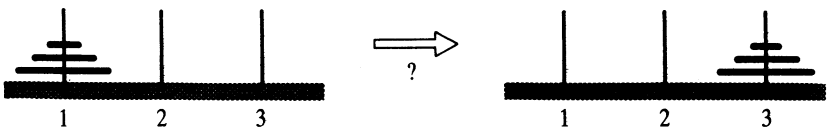


Figure 13.6 The Tower of Hanoi problem.

But the first goal cannot immediately be achieved: disk c cannot move in the initial situation. Therefore, we first have to prepare this move and our strategy is refined to:

(1) Enable moving disk c from 1 to 3.
(2) Move disk c from 1 to 3.
(3) Achieve remaining goals: a on 3, and b on 3.

Disk c can only move from 1 to 3 if both a and b are stacked on peg 2. Then, our initial problem of moving a, b and c from peg 1 to peg 3 is reduced to three subproblems:

To move a, b and c from 1 to 3:
(1) move a and b from 1 to 2, *and*
(2) move c from 1 to 3, *and*
(3) move a and b from 2 to 3.

Problem 2 is trivial (one-step solution). The other two subproblems can be solved independently of problem 2 because disks a and b can be moved regardless of the position of disk c. To solve problems 1 and 3, the same decomposition principle can be applied (disk b is the hardest this time). Accordingly, problem 1 is reduced to three trivial subproblems:

To move a and b from 1 to 2:
(1) move a from 1 to 3, *and*
(2) move b from 1 to 2, *and*
(3) move a from 3 to 2.

13.2.3 AND/OR formulation of game-playing

Games like chess and checkers can naturally be viewed as problems, represented by AND/OR graphs. Such games are called two-person, perfect-information games, and we will assume here that there are only two possible outcomes: WIN or LOSS. (We can think of games with three outcomes – WIN, LOSS and DRAW – as also having just two outcomes: WIN and NO-WIN.) As the two players move in turn we have two kinds of positions, depending on who is to move. Let us call the two players 'us' and 'them', so the two kinds of positions are: 'us-to-move' positions and 'them-to-move' positions. Assume that the game starts in an us-to-move position P. Each alternative us-move in this position leads to one of them-to-move positions Q_1, Q_2, ... (Figure 13.7). Further, each alternative them-move in Q_1 leads to one of the positions R_{11}, R_{12}, ... In the AND/OR tree of Figure 13.7, nodes correspond to positions, and arcs correspond to possible moves. Us-to-move levels alternate with them-to-move levels. To win in the initial position, P, we have to find a move from P

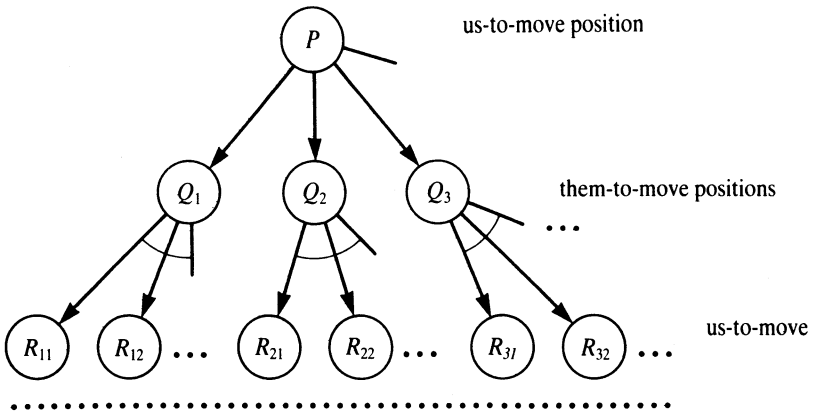


Figure 13.7 An AND/OR formulation of a two-person game; the players are 'us' and 'them'.

to Q_i, for some i, so that the position Q_i is won. Thus, P is won if Q_1 *or* Q_2 *or* ... is won. Therefore position P is an OR node. For all i, position Q_i is a them-to-move position, so if it is to be won for us it has to be won after *each* them-move. Thus Q_i is won if all R_{i1} *and* R_{i2} *and* ... are won. Accordingly, all them-to-move positions are AND nodes. Goal nodes are those positions that are won by the rules of the game; for example, their king checkmated in chess. Those positions that are lost by the rules of the game correspond to unsolvable problems. To solve the game we have to find a solution tree that guarantees our victory regardless of the opponent's replies. Such a solution tree, then, is a complete strategy for winning the game: for each possible continuation that can be chosen by the opponent, there is a move in such a strategy tree that forces a win.

13.3 Basic AND/OR search procedures

In this section we will only be interested in finding *some* solution of the problem, regardless of its cost. So for the purposes of this section we can ignore the costs of links or nodes in an AND/OR graph.

The simplest way of searching AND/OR graphs in Prolog is to use Prolog's own search mechanism. This happens to be trivial as Prolog's procedural meaning itself is nothing but a procedure for searching AND/OR graphs. For example, the AND/OR graph of Figure 13.4 (ignoring the arc costs) can be specified by the following clauses:

```
a :- b.          % a is an OR node with two successors, b and c
a :- c.
b :- d, e.       % b is an AND node with two successors, d and e
```

```
e :- h.

c :- f, g.

f :- h, i.

d.      g.      h.   % d, g and h are goal nodes
```

To ask whether problem **a** can be solved we can simply ask:

```
?- a.
```

Now Prolog will effectively search the tree of Figure 13.4 in the depth-first fashion and answer 'yes', after having visited that part of the search graph corresponding to the solution tree in Figure 13.4(b).

The advantage of this approach to programming AND/OR search is its simplicity. There are disadvantages, however:

- We only get an answer 'yes' or 'no', not a solution tree as well. We could reconstruct the solution tree from the program's trace, but this can be awkward and insufficient if we want a solution tree explicitly accessible as an object in the program.
- This program is hard to extend so as to be able to handle costs as well.
- If our AND/OR graph were a general graph, containing cycles, then Prolog with its depth-first strategy could enter an indefinite recursive loop.

Let us rectify these deficiencies gradually. We will first define our own depth-first search procedure for AND/OR graphs.

To begin with, we have to change the representation of AND/OR graphs in the program. For that we will introduce a binary relation represented in the infix notation with the operator '--->'. For example, node *a* linked to its two OR successors will be represented by the clause:

```
a ---> or : [b,c].
```

The symbols '--->' and ':' are both infix operators that can be defined by:

```
:- op( 600, xfx, --->).

:- op( 500, xfx, :).
```

The complete AND/OR graph of Figure 13.4 is thus specified by the clauses:

```
a  --->  or : [b,c].
b  --->  and : [d,e].
c  --->  and : [f,g].
e  --->  or : [h].
```

```
f  --->  or : [h,i].

goal( d).      goal( g).      goal( h).
```

The depth-first AND/OR procedure can be constructed from the following principles:

> To solve a node, N, use the following rules:
>
> (1) If N is a goal node then it is trivially solved.
>
> (2) If N has OR successors then solve one of them (attempt them one after another until a solvable one is found).
>
> (3) If N has AND successors then solve all of them (attempt them one after another until they have all been solved).
>
> If the above rules do not produce a solution then assume the problem cannot be solved.

A corresponding program can be as follows:

```
solve( Node) :-
  goal( Node).

solve( Node) :-
  Node ---> or : Nodes,          % Node is an OR node
  member( Node1, Nodes),         % Select a successor Node1 of Node
  solve( Node1).

solve( Node) :-
  Node ---> and : Nodes,         % Node is an AND node
  solveall( Nodes).              % Solve all Node's successors

solveall( [] ).

solveall( [Node | Nodes] ) :-
  solve( Node),
  solveall( Nodes).
```

member is the usual list membership relation.

This program still has the following disadvantages:

- it does not produce a solution tree, and
- it is susceptible to infinite loops, depending on the properties of the AND/OR graph (cycles).

The program can easily be modified to produce a solution tree. We have to modify the **solve** relation so that it has two arguments:

```
solve( Node, SolutionTree)
```

Let us represent a solution tree as follows. We have three cases:

(1) If **Node** is a goal node then the corresponding solution tree is **Node** itself.

(2) If **Node** is an OR node then its solution tree has the form:

Node ---> Subtree

where **Subtree** is a solution tree for one of the successors of **Node**.

(3) If **Node** is an AND node then its solution tree has the form:

Node ---> and : Subtrees

where **Subtrees** is the list of solution trees of all of the successors of **Node**.

For example, in the AND/OR graph of Figure 13.4, the first solution of the top node *a* is represented by:

a ---> b ---> and : [d, e ---> h]

The three forms of a solution tree correspond to the three clauses about our **solve** relation. So our initial **solve** program can be altered by simply modifying each of the three clauses; that is, by just adding solution tree as the second argument to **solve**. The resulting program is shown as Figure 13.8. An additional procedure in this program is **show** for displaying solution trees. For example, the solution tree of Figure 13.4 is displayed by **show** in the following form:

```
a ---> b ---> d
               e ---> h
```

The program of Figure 13.8 is still prone to infinite loops. One simple way to prevent infinite loops is to keep trace of the current depth of the search and prevent the program from searching beyond some depth limit. We can do this by simply introducing another argument to the **solve** relation:

solve(Node, SolutionTree, MaxDepth)

As before, **Node** represents a problem to be solved, and **SolutionTree** is a solution not deeper than **MaxDepth**. **MaxDepth** is the allowed depth of search in the graph. In the case that **MaxDepth** = 0 no further expansion is allowed; otherwise, if **MaxDepth** > 0 then **Node** can be expanded and its successors are attempted with a lower depth limit **MaxDepth** – 1. This can easily be incorporated into the program of Figure 13.8. For example, the second clause about **solve** becomes:

```
solve( Node, Node ---> Tree, MaxDepth) :-
  MaxDepth > 0,
  Node ---> or : Nodes,              % Node is an OR node
  member( Node1, Nodes),             % Select a successor Node1 of Node
  Depth1 is MaxDepth - 1,            % New depth limit
  solve( Node1, Tree, Depth1).       % Solve successor with lower limit
```

This depth-limited, depth-first procedure can also be used for simulating the breadth-first search. The idea is to do the depth-first search repetitively, each time with a greater depth limit, until a solution is found. That is, try to

```
% Depth-first AND/OR search
% Procedure solve( Node, SolutionTree) finds a solution tree
% for a node in an AND/OR graph

solve( Node, Node) :-                    % Solution tree of goal node is Node itself
  goal( Node).

solve( Node, Node ---> Tree) :-
  Node ---> or : Nodes,                  % Node is an OR node
  member( Node1, Nodes),                 % Select a successor Node1 of Node
  solve( Node1, Tree).

solve( Node, Node ---> and : Trees) :-
  Node ---> and : Nodes,                 % Node is an AND node
  solveall( Nodes, Trees).               % Solve all Node's successors

solveall( [], [] ).

solveall( [Node | Nodes], [Tree | Trees] ) :-
  solve( Node, Tree),
  solveall( Nodes, Trees).

show( Tree) :-                           % Display solution tree
  show( Tree, 0), !.                     % Indented by 0

show( Node ---> Tree, H) :-              % Display solution tree indented by H
  write( Node), write( ' ---> '),
  H1 is H + 7,
  show( Tree, H1), !.

show( and : [T], H) :-                   % Display AND list of solution trees
  show( T, H).

show( and : [T | Ts], H) :-              % Display AND list of solution trees
  show( T, H),
  tab( H),
  show( and : Ts, H), !.

show( Node, H) :-
  write( Node), nl.
```

Figure 13.8 Depth-first search for AND/OR graphs. This program does not avoid infinite cycling. Procedure **solve** finds a solution tree and procedure **show** displays such a tree. **show** assumes that each node only takes one character on output.

solve the problem with depth limit 0, then with 1, then with 2, etc. Such a program is:

```
simulated_breadth_first( Node, SolTree) :-
  trydepths( Node, SolTree, 0).          % Try search with increasing
                                         % depth limit, start with 0

trydepths( Node, SolTree, Depth) :-
  solve( Node, SolTree, Depth);
  Depth1 is Depth + 1,                   % Get new depth limit
  trydepths( Node, SolTree, Depth1).     % Try higher depth limit
```

A disadvantage of this breadth-first simulation is that the program researches top parts of the search space each time that the depth limit is increased.

Exercises

13.1 Complete the depth-limited, depth-first AND/OR search program according to the procedure outlined in this section.

13.2 Define in Prolog an AND/OR space for the Tower of Hanoi problem and use this definition with the search procedures of this section.

13.3 Consider some simple two-person, perfect-information game without chance and define its AND/OR representation. Use a depth-first AND/OR search program to find winning strategies in the form of AND/OR trees.

13.4 Best-first AND/OR search

13.4.1 Heuristic estimates and the search algorithm

The basic search procedures of the previous section search AND/OR graphs systematically and exhaustively, without any heuristic guidance. For complex problems such procedures are too inefficient due to the combinatorial complexity of the search space. Heuristic guidance that aims to reduce the complexity by avoiding useless alternatives becomes necessary. The heuristic guidance introduced in this section will be based on numerical heuristic estimates of the difficulty of problems in the AND/OR graph. The program that we shall develop can be viewed as a generalization of the best-first search program for the state-space representation of Chapter 12.

Let us begin by introducing an optimization criterion based on the costs of arcs in the AND/OR graph. First, we extend our representation of AND/OR

graphs to include arc costs. For example, the AND/OR graph of Figure 13.4 can be represented by the following clauses:

```
a ---> or : [b/1, c/3].
b ---> and : [d/1, e/1].
c ---> and : [f/2, g/1].
e ---> or : [h/6].
f ---> or : [h/2, i/3].

goal( d).    goal( g).    goal( h).
```

We shall define the cost of a solution tree as the sum of all the arc costs in the tree. The optimization objective is to find a minimum-cost solution-tree. For illustration, see Figure 13.4 again.

It is useful to define the *cost of a node* in the AND/OR graph as the cost of the node's optimal solution tree. So defined, the cost of a node corresponds to the difficulty of the node.

We shall now assume that we can estimate the costs of nodes (without knowing their solution trees) in the AND/OR graph with some heuristic function h. Such estimates will be used for guiding the search. Our heuristic search program will begin the search with the start node and, by expanding already visited nodes, gradually grow a search tree. This process will grow a *tree* even in cases that the AND/OR graph itself is not a tree; in such a case the graph unfolds into a tree by duplicating parts of the graph.

The search process will at any time of the search select the 'most promising' candidate solution tree for the next expansion. Now, how is the function h used to estimate how promising a candidate solution tree is? Or, how promising a node (the root of a candidate solution tree) is?

For a node N in the search tree, $H(N)$ will denote its estimated difficulty. For a tip node N of the current search tree, $H(N)$ is simply $h(N)$. On the other hand, for an interior node of the search tree we do not have to use function h directly because we already have some additional information about such a node; that is, we already know its successors. Therefore, as Figure 13.9 shows, for an interior OR node N we approximate its difficulty as:

$$H(N) = \min_i (cost(N,N_i) + H(N_i))$$

where $cost(N,N_i)$ is the cost of the arc from N to N_i. The minimization rule in this formula is justified by the fact that, to solve N, we just have to solve one of its successors.

The difficulty of an AND node N is approximated by:

$$H(N) = \sum_i (cost(N,N_i) + H(N_i))$$

We say that the H-value of an interior node is a 'backed-up' estimate.

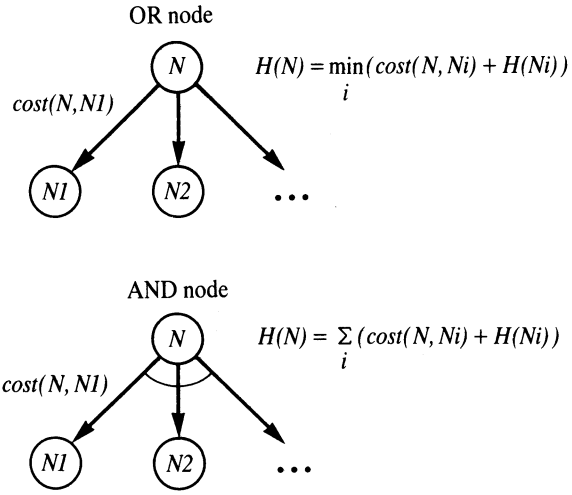


Figure 13.9 Estimating the difficulty, H, of problems in the AND/OR graph.

In our search program, it will be more practical to use (instead of the H-values) another measure, F, defined in terms of H, as follows. Let a node M be the predecessor of N in the search tree, and the cost of the arc from M to N be $cost(M,N)$, then we define:

$$F(N) = cost(M,N) + H(N)$$

Accordingly, if M is the parent node of N, and $N_1, N_2, \ldots$ are N's children, then:

$$F(N) = cost(M,N) + \min_i F(N_i), \qquad \text{if } N \text{ is an OR node}$$

$$F(N) = cost(M,N) + \sum_i F(N_i), \qquad \text{if } N \text{ is an AND node}$$

The start node S of the search has no predecessor, but let us choose the cost of its (virtual) incoming arc as 0. Now, if h for all goal nodes in the AND/OR graph is 0, and an optimal solution tree has been found, then $F(S)$ is just the cost of this solution tree (that is, the sum of all the costs of its arcs).

At any stage of the search, each successor of an OR node represents an alternative candidate solution subtree. The search process will always decide to continue the exploration at that successor whose F-value is minimal. Let us return to Figure 13.4 again and trace such a search process when searching the AND/OR graph of this figure. Initially, the search tree is just the start node a, and then the tree grows until a solution tree is found. Figure 13.10 shows some snapshots taken during the growth of the search tree. We shall assume for simplicity that $h = 0$ for all the nodes. Numbers attached to nodes in Figure 13.10 are the F-values of the nodes (of course, these change during the search

as more information is accumulated). Here are some explanatory remarks to Figure 13.10.

Expanding the initial search tree (snapshot A) produces tree B. Node *a* is an OR node, so we now have two candidate solution trees: *b* and *c*. As

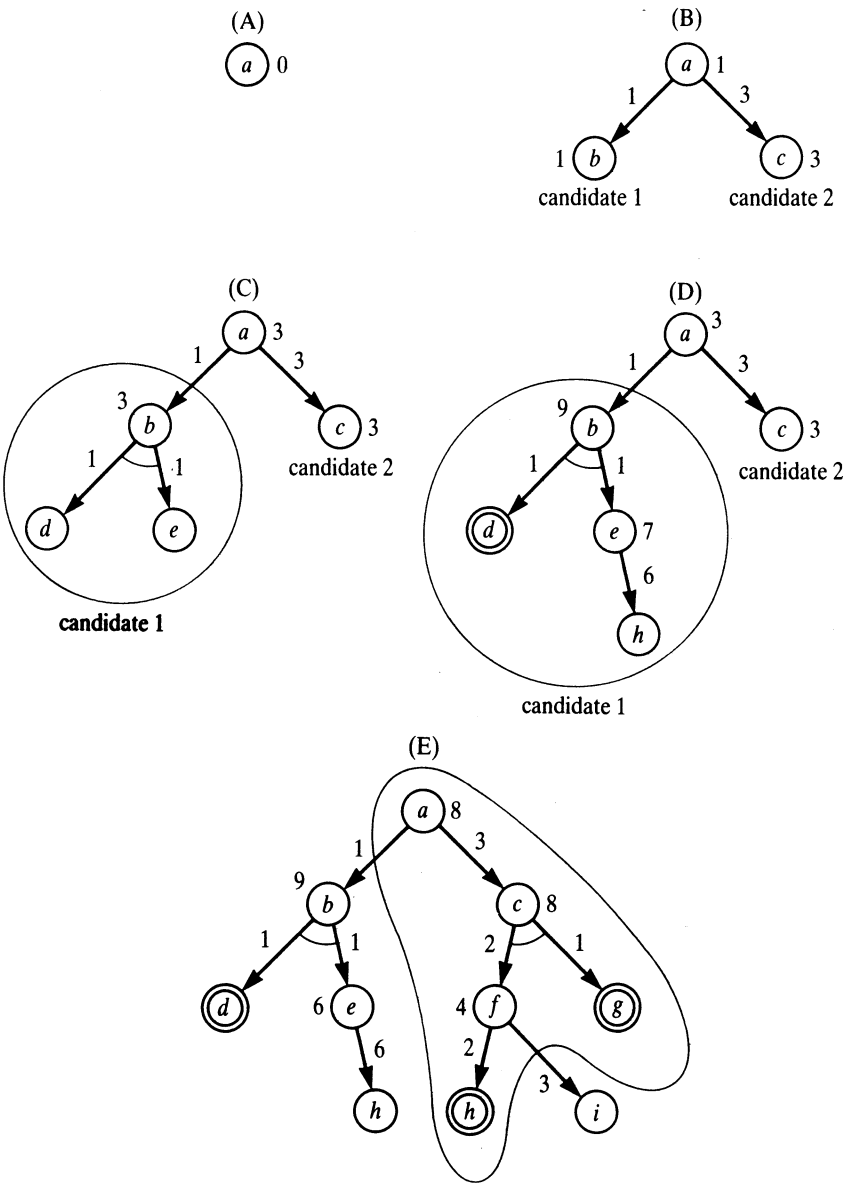


Figure 13.10 A trace of a best-first AND/OR search (using $h = 0$) solving the problem of Figure 13.4.

$F(b) = 1 < 3 = F(c)$, alternative b is selected for expansion. Now, how far can alternative b be expanded? The expansion can proceed until either:

(1) the F-value of node b has become greater than that of its competitor c, or
(2) it has become clear that a solution tree has been found.

So candidate b starts to grow with the upper bound for $F(b)$: $F(b) \leqq 3 = F(c)$. First, b's successors d and e are generated (snapshot C) and the F-value of b is increased to 3. As this does not exceed the upper bound, the candidate tree rooted in b continues to expand. Node d is found to be a goal node, and then node e is expanded resulting in snapshot D. At this point $F(b) = 9 > 3$ which stops the expansion of alternative b. This prevents the process from realizing that h is also a goal node and that a solution tree has already been generated. Instead, the activity now switches to the competing alternative c. The bound on $F(c)$ for expanding this alternative is set to 9, since at this point $F(b) = 9$. Within this bound the candidate tree rooted in c is expanded until the situation of snapshot E is reached. Now the process realizes that a solution tree (which includes goal nodes h and g) has been found, and the whole process terminates. Notice that the cheaper of the two possible solution trees was reported a solution by this process – that is, the solution tree in Figure 13.4(c).

13.4.2 Search program

A program that implements the ideas of the previous section is given in Figure 13.12. Before explaining some details of this program, let us consider the representation of the search tree that this program uses.

There are several cases, as shown in Figure 13.11. The different forms of the search tree arise from combining the following possibilities with respect to the tree's size and 'solution status':

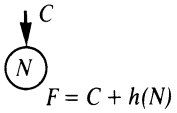
- Size:
  (1) the tree is either a single node tree (a leaf), or
  (2) it has a root and (non-empty) subtrees.
- Solution status:
  (1) the tree has already been discovered to be solved (the tree is a solution tree), or
  (2) it is still just a *candidate* solution tree.

The principal functor used to represent the tree indicates a combination of these possibilities. This can be one of the following:

leaf **solvedleaf** **tree** **solvedtree**

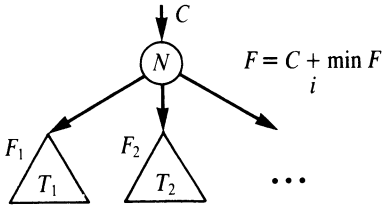
Further, the representation comprises some or all of the following information:

Case 1: Search leaf



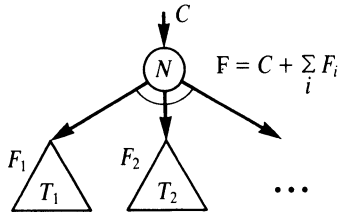
leaf(N,F,C)

Case 2: Search tree with OR subtrees



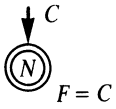
tree(N,F,C,or:[T1,T2,...])

Case 3: Search tree with AND subtrees



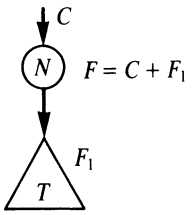
tree(N,F,C,and:[T1,T2,...])

Case 4: Solved leaf



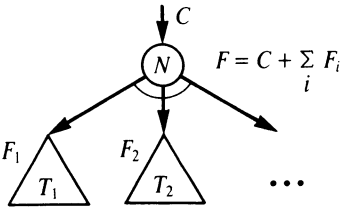
solvedleaf (N, F)

Case 5: Solution tree rooted at OR node



solvedtree(N,F,T)

Case 6: Solution tree rooted at AND node



solvedtree(N,F,and:[T1,T2,...])

Figure 13.11 Representation of the search tree.

- root node of the tree,
- *F*-value of the tree,
- the cost C of the arc in the AND/OR graph pointing to the tree,
- list of subtrees,
- relation among subtrees (AND or OR).

The list of subtrees is always ordered according to increasing *F*-values. A subtree can already be solved. Such subtrees are accommodated at the end of the list.

Now to the program of Figure 13.12. The top-level relation is

andor(Node, SolutionTree)

where **Node** is the start node of the search. The program produces a solution tree (if one exists) with the aspiration that this will be an optimal solution. Whether it will really be a cheapest solution depends on the heuristic function *h* used by the algorithm. There is a theorem that talks about this dependance on *h*. The theorem is similar to the admissibility theorem about the state-space, best-first search of Chapter 12. Let *COST(N)* denote the cost of a cheapest solution tree of a node *N*. If for each node *N* in the AND/OR graph the heuristic estimate $h(N) \leqq COST(N)$ then **andor** is guaranteed to find an optimal solution. If *h* does not satisfy this condition then the solution found may be suboptimal. A trivial heuristic function that satisfies the admissibility condition is $h = 0$ for all the nodes. The disadvantage of this function is, of course, lack of heuristic power.

The key relation in the program of Figure 13.12 is

expand(Tree, Bound, Tree1, Solved)

Tree and **Bound** are 'input' arguments, and **Tree1** and **Solved** are 'output' arguments. Their meaning is:

Tree is a search tree that is to be expanded.
Bound is a limit for the *F*-value within which **Tree** is allowed to expand.
Solved is an indicator whose value indicates one of the following three cases:

(1) **Solved** = **yes**: **Tree** can be expanded within bound so as to comprise a solution tree **Tree1**;

(2) **Solved** = **no**: **Tree** can be expanded to **Tree1** so that the *F*-value of **Tree1** exceeds **Bound**, and there was no solution subtree before the *F*-value overstepped **Bound**;

(3) **Solved** = **never**: **Tree** is unsolvable.

Tree1 is, depending on the cases above, either a solution tree, an extension of **Tree** just beyond **Bound**, or uninstantiated in the case **Solved** = **never**.

```
/* BEST-FIRST AND/OR SEARCH

This program only generates one solution. This solution is guaranteed to be a
cheapest one if the heuristic function used is a lower bound of the actual costs of
solution trees.

Search tree is either:
tree( Node, F, C, SubTrees)          tree of candidate solutions
leaf( Node, F, C)                    leaf of a search tree
solvedtree( Node, F, SubTrees)       solution tree
solvedleaf( Node, F)                 leaf of solution tree
C is the cost of the arc pointing to Node
F = C + H, where H is the heuristic estimate of an optimal solution subtree rooted in
          Node
SubTrees are always ordered so that:
(1) all solved subtrees are at the end of a list;
(2) other (unsolved subtrees) are ordered according to ascending F-values.
*/


:- op( 500, xfx, :).

:- op( 600, xfx, --->).

andor( Node, SolutionTree) :-
  expand( leaf( Node, 0, 0), 9999, SolutionTree, yes).  % Assuming 9999 < any
                                                        % F-value

% Procedure expand( Tree, Bound, NewTree, Solved)
% expands Tree with Bound producing NewTree whose
% 'solution-status' is Solved
% Case 1: bound exceeded

expand( Tree, Bound, Tree, no) :-
  f( Tree, F), F > Bound, !.                            % Bound exceeded


% In all remaining cases F ≦ Bound
% Case 2: goal encountered

expand( leaf( Node, F, C), _, solvedleaf( Node, F), yes) :-
  goal( Node), !.


% Case 3: expanding a leaf

expand( leaf( Node, F, C), Bound, NewTree, Solved) :-
  expandnode( Node, C, Tree1), !,
  expand( Tree1, Bound, NewTree, Solved);
  Solved = never, !.                                    % No successors, dead end
```

```
% Case 4: expanding a tree

expand( tree( Node, F, C, SubTrees), Bound, NewTree, Solved) :-
  Bound1 is Bound - C,
  expandlist( SubTrees, Bound1, NewSubs, Solved1),
  continue( Solved1, Node, C, NewSubs, Bound, NewTree, Solved).

% expandlist( Trees, Bound, NewTrees, Solved)
% expands tree list Trees with Bound producing
% NewTrees whose 'solved-status' is Solved

expandlist( Trees, Bound, NewTrees, Solved) :-
  selecttree( Trees, Tree, OtherTrees, Bound, Bound1),
  expand( Tree, Bound1, NewTree, Solved1),
  combine( OtherTrees, NewTree, Solved1, NewTrees, Solved).

% 'continue' decides how to continue after expanding a tree list

continue( yes, Node, C, SubTrees, _, solvedtree( Node, F, SubTrees), yes) :-
  backup( SubTrees, H), F is C + H, !.

continue( never, _, _, _, _, _, never) :- !.

continue( no, Node, C, SubTrees, Bound, NewTree, Solved) :-
  backup( SubTrees, H), F is C + H, !,
  expand( tree( Node, F, C, SubTrees), Bound, NewTree, Solved).

% 'combine' combines results of expanding a tree and a tree list

combine( or : _, Tree, yes, Tree, yes) :- !.          % OR list solved

combine( or : Trees, Tree, no, or : NewTrees, no) :-
  insert( Tree, Trees, NewTrees), !.                  % OR list still unsolved

combine( or : [], _, never, _, never) :- !.           % No more candidates

combine( or : Trees, _, never, or : Trees, no) :- !. % There are more candidates.

combine( and : Trees, Tree, yes, and : [Tree | Trees], yes) :-
  allsolved( Trees), !.                               % AND list solved

combine( and : _, _, never, _, never) :- !.           % AND list unsolvable

combine( and : Trees, Tree, YesNo, and : NewTrees, no) :-
  insert( Tree, Trees, NewTrees), !.                  % AND list still unsolved

% 'expandnode' makes a tree of a node and its successors

expandnode( Node, C, tree( Node, F, C, Op : SubTrees) ) :-
  Node ---> Op : Successors,
  evaluate( Successors, SubTrees),
  backup( Op : SubTrees, H), F is C + H.

evaluate( [], [] ).
```

```
evaluate( [Node/C | NodesCosts], Trees) :-
  h( Node, H), F is C + H,
  evaluate( NodesCosts, Trees1),
  insert( leaf( Node, F, C), Trees1, Trees).

% 'allsolved' checks whether all trees in a tree list are solved

allsolved( [] ).

allsolved( [Tree | Trees] ) :-
  solved( Tree),
  allsolved( Trees).

solved( solvedtree( _, _, _) ).

solved( solvedleaf( _, _) ).

f( Tree, F) :-                          % Extract F-value of a tree
  arg( 2, Tree, F), !.                  % F is the 2nd argument in Tree

% insert( Tree, Trees, NewTrees) inserts Tree into
% tree list Trees producing NewTrees

insert( T, [], [T] ) :- !.

insert( T, [T1 | Ts], [T, T1 | Ts] ) :-
  solved( T1), !.

insert( T, [T1 | Ts], [T1 | Ts1] ) :-
  solved( T),
  insert( T, Ts, Ts1), !.

insert( T, [T1 | Ts], [T, T1 | Ts ] ) :-
  f( T, F), f( T1, F1), F =< F1, !.

insert( T, [T1 | Ts], [T1 | Ts1] ) :-
  insert( T, Ts, Ts1).

% 'backup' finds the backed-up F-value of AND/OR tree list

backup( or : [Tree | _], F) :-          % First tree in OR list is best
  f( Tree, F), !.

backup( and : [], 0) :- !.

backup( and : [Tree1 | Trees], F) :-
  f( Tree1, F1),
  backup( and : Trees, F2),
  F is F1 + F2, !.

backup( Tree, F) :-
  f( Tree, F).
```

```
% Relation selecttree( Trees, BestTree, OtherTrees, Bound, Bound1):
% OtherTrees is an AND/OR list Trees without its best member
% BestTree; Bound is expansion bound for Trees, Bound1 is
% expansion bound for BestTree

selecttree( Op : [Tree], Tree, Op : [], Bound, Bound) :- !. % The only candidate

selecttree( Op : [Tree | Trees], Tree, Op : Trees, Bound, Bound1) :-
  backup( Op : Trees, F),
  ( Op = or, !, min( Bound, F, Bound1);
    Op = and, Bound1 is Bound - F).

min( A, B, A) :- A < B, !.

min( A, B, B).
```

Figure 13.12 Best-first AND/OR search program.

Procedure

expandlist(Trees, Bound, Trees1, Solved)

is similar to **expand**. As in **expand**, **Bound** is a limit of the expansion of a tree, and **Solved** is an indicator of what happened during the expansion ('yes', 'no' or 'never'). The first argument is, however, a *list* of trees (an AND list or an OR list):

Trees = or : [T1, T2, ...] or **Trees = and : [T1, T2, ...]**

expandlist selects the most promising tree T (according to *F*-values) in **Trees**. Due to the ordering of the subtrees this is always the first tree in **Trees**. This most promising subtree is expanded with a new bound **Bound1**. **Bound1** depends on **Bound** and also on the other trees in **Trees**. If **Trees** is an OR list then **Bound1** is the lower of **Bound** and the *F*-value of the next best tree in **Trees**. If **Trees** is an AND list then **Bound1** is **Bound** minus the sum of the *F*-values of the remaining trees in **Trees**. **Trees1** depends on the case indicated by **Solved**. In the case **Solved = no**, **Trees1** is the list **Trees** with the most promising tree in **Trees** expanded with **Bound1**. In the case **Solved = yes**, **Trees1** is a solution of the list **Trees** (found within **Bound**). If **Solved = never**, **Trees1** is uninstantiated.

The procedure **continue**, which is called after expanding a tree list, decides what to do next, depending on the results of **expandlist**. It either constructs a solution tree, or updates the search tree and continues its expansion, or signals 'never' in the case that the tree list was found unsolvable.

Another procedure,

combine(OtherTrees, NewTree, Solved1, NewTrees, Solved)

relates several objects dealt with in **expandlist**. **NewTree** is the expanded tree in the tree list of **expandlist**, **OtherTrees** are the remaining, unchanged trees in the tree list, and **Solved1** indicates the 'solution-status' of **NewTree**. **combine** handles several cases, depending on **Solved1** and on whether the tree list is an AND list or an OR list. For example, the clause

```
combine( or : _, Tree, yes, Tree, yes).
```

says: in case that the tree list is an OR list, and the just expanded tree was solved, and its solution tree is **Tree**, then the whole list has also been solved, and its solution is **Tree** itself. Other cases are best understood from the code of **combine** itself.

For displaying a solution tree, a procedure similar to **show** of Figure 13.8 can be defined. This procedure is left as an exercise for the reader.

13.4.3 Example of problem-defining relations: route finding

Let us now formulate the route-finding problem as an AND/OR search so that this formulation can be directly used by our **andor** procedure of Figure 13.12. We shall assume that the road map is represented by a relation

```
s( City1, City2, D)
```

meaning that there is a direct connection between **City1** and **City2** of distance D. Further, we shall assume there is a relation

```
key( City1 - City2, City3)
```

meaning: to find a route from **City1** to **City2** we should consider paths that go through **City3** (**City3** is a 'key point' between **City1** and **City2**). For example, in the map of Figure 13.1, *f* and *g* are key points between *a* and *z*:

```
key( a-z, f).      key( a-z, g).
```

We shall implement the following principles of route finding:

To find a route between two cities X and Z:

(1) if there are key points Y1, Y2, ... between X and Z then find either
- route from A to Z via Y1, or
- route from A to Z via Y2, or

...

(2) if there is no key point between X and Z then simply find some neighbour city Y of X such that there is a route from Y to Z.

We have, then, two kinds of problems that will be represented as:

(1) **X-Z** find a route from X to Z
(2) **X-Z via Y** find a route from X to Z through Y

Here 'via' is an infix operator with precedence higher than that of '-' and lower than that of '--->'. The corresponding AND/OR graph can now be implicitly defined by the following piece of program:

```
:- op( 560, xfx, via).

% Expansion rule for problem X-Z when
% there are key points between X and Z,
% costs of all arcs are equal 0

X-Z ---> or : ProblemList
  :- bagof( ( X-Z via Y)/0, key( X-Z, Y), ProblemList), !.

% Expansion rule for problem X-Z with no key points

X-Z ---> or : ProblemList
  :- bagof( ( Y-Z)/D, s( X, Y, D), ProblemList).

% Reduce a 'via problem' to two AND-related subproblems

X-Z via Y ---> and : [( X-Y)/0, ( Y-Z)/0].

goal( X-X).                          % To go from X to X is trivial
```

The function *h* could be defined, for example, as the air distance between cities.

Exercise

13.4 Write a procedure

show2(SolutionTree)

to display a solution tree found by the **andor** program of Figure 13.12. Let the display format be similar to that of the **show** procedure of Figure 13.8, so that **show2** can be written as a modification of **show**, using a different tree representation. Another useful modification would be to replace the goal **write(Node)** in **show** by a user-defined procedure

writenode(Node, H)

which outputs **Node** in some suitable form, and instantiates H to the number of characters that **Node** takes if output in this form. H is then used for proper indentation of subtrees.

Summary

- AND/OR graph is a formalism for representing problems. It naturally suits problems that are decomposable into independent subproblems. Game playing is an example of such problems.
- Nodes of an AND/OR graph are of two types: AND nodes and OR nodes.
- A concrete problem is defined by a start node and a goal condition. A solution of a problem is represented by a solution graph.
- Costs of arcs and nodes can be introduced into an AND/OR graph to model optimization problems.
- Solving a problem, represented by an AND/OR graph, involves searching the graph. The depth-first strategy searches the graph systematically and is easy to program. However, it may suffer from inefficiency due to combinatorial explosion.
- Heuristic estimates can be introduced to estimate the difficulty of problems, and the best-first heuristic principle can be used to guide the search. Implementing this strategy is more difficult.
- Prolog programs for depth-first search and best-first search of AND/OR graphs were developed in this chapter.
- Concepts introduced in this chapter are:

  AND/OR graphs
  AND arcs, OR arcs
  AND nodes, OR nodes
  solution graph, solution tree
  arc costs, node costs
  heuristic estimates in AND/OR graphs, backed-up estimates
  depth-first search in AND/OR graphs
  best-first search in AND/OR graphs

References

AND/OR graphs and related search algorithms are part of the classical Artificial Intelligence problem-solving and game-playing machinery. An early example of their application is a symbolic integration program (Slagle 1963). Prolog itself does AND/OR search. A general treatment of AND/OR graphs and the best-first AND/OR search algorithm can be found in general books on Artificial Intelligence (Nilsson 1971; Nilsson 1980). Our best-first AND/OR program is a variation of an algorithm known as AO*. Formal properties of AO* (including its admissibility) have been studied by several authors; Pearl (1984) gives a comprehensive account of these results.

Nilsson, N. J. (1971) *Problem-Solving Methods in Artificial Intelligence*. McGraw-Hill.

Nilsson, N. J. (1980) *Principles of Artificial Intelligence*. Tioga; also Springer-Verlag.

Pearl, J. (1984) *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley.

Slagle, J. R. (1963) A heuristic program that solves symbolic integration problems in freshman calculus. In: *Computers and Thought* (E. Feigenbaum, J. Feldman, eds.). McGraw-Hill.

14 Expert Systems

An expert system is a program that behaves like an expert for some problem domain. It should be capable of *explaining* its decisions and the underlying reasoning. Often an expert system is expected to be able to deal with uncertain and incomplete information.

To build an expert system we have to develop the following functions: problem-solving function, user-interaction function, and dealing with uncertainty. In this chapter we will develop and implement a framework of basic ideas for building expert systems.

14.1 Functions of an expert system

An *expert system* is a program that behaves like an expert in some, usually narrow, domain of application. Typical applications include tasks such as medical diagnosis, locating a failure in certain kind of equipment, or interpreting measurement data. Expert systems have to be capable of solving problems that require expert knowledge in a particular domain. They should possess that knowledge in some form. Therefore they are also called *knowledge-based systems*. However, not every knowledge-based system can be considered an expert system. An expert system also has to be capable, in some way, of *explaining* its behaviour and its decisions to the user, as human experts do. Such an explanation feature is especially necessary in uncertain domains (such as medical diagnosis) in order to enhance the user's confidence in the system's advice, or to enable the user to detect a possible flaw in the system's reasoning. Therefore, expert systems have to have a friendly user-interaction capability that will make the system's reasoning transparent to the user.

An additional feature that is often required of an expert system is the ability to deal with uncertainty and incompleteness. Information about the problem to be solved can be incomplete or unreliable; relations in the problem domain can be approximate. For example, we may not be quite sure that some symptom is present in the patient, or that some measurement data is absolutely correct; some drug *may* cause some problem, but *usually* does not. All this requires probabilistic reasoning.

To build an expert system we have, in general, to develop the following functions:

- *problem-solving* function capable of using domain-specific knowledge – this may require *dealing with uncertainty*
- *user-interaction* function, which includes explanation of the system's intentions and decisions during and after the problem-solving process.

Each of these functions can be very complicated, and can depend on the domain of application and practical requirements. Various intricate problems may arise in the design and implementation. In this chapter we will develop a framework of basic ideas that can be further refined.

14.2 Main structure of an expert system

It is convenient to divide the development of an expert system into three main modules, as illustrated in Figure 14.1:

(1) a knowledge base,
(2) an inference engine,
(3) a user interface.

A *knowledge-base* comprises the knowledge that is specific to the domain of application, including such things as simple facts about the domain, rules that describe relations or phenomena in the domain, and possibly also methods, heuristics and ideas for solving problems in this domain. An *inference engine* knows how to actively use the knowledge in the base. A *user interface* caters for smooth communication between the user and the system, also providing the user with an insight into the problem-solving process carried out by the inference engine. It is convenient to view the inference engine and the interface as one module, usually called an *expert system shell*, or simply a *shell* for brevity.

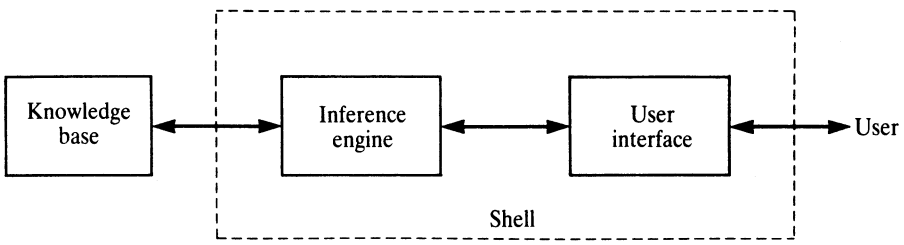


Figure 14.1 Structure of expert systems.

The foregoing scheme separates knowledge from algorithms that use the knowledge. This division is suitable for the following reasons: the knowledge base clearly depends on the application. On the other hand, the shell is, in principle at least, domain independent. Thus a rational way of developing expert systems for several applications consists of developing a shell that can be used universally, and then to plug in a new knowledge base for each application. Of course, all the knowledge bases will have to conform to the same formalism that is 'understood' by the shell. According to practical experience in complex expert systems the scenario with one shell and many knowledge bases will not work quite so smoothly unless the application domains are indeed very similar. Nevertheless, even if modifications of the shell from one domain to another are necessary, at least the main principles can be retained.

In this chapter we are going to develop a comparatively simple expert system shell which will, despite its simplicity, illustrate fundamental ideas and techniques of this field. Our development plan will be as follows:

(1) Select a formalism for representing knowledge.
(2) Design an inference mechanism that corresponds to this formalism.
(3) Add user-interaction facilities.
(4) Add a facility for handling uncertainty.

14.3 *If-then* rules for representing knowledge

In principle, any consistent formalism in which we can express knowledge about some problem domain can be considered for use in an expert system. However, the language of *if-then* rules, also called *production* rules, is by far the most popular formalism for representing knowledge and will be used here. In general, such rules are conditional statements, but they can have various interpretations. Examples are:

- *if* precondition P *then* conclusion C
- *if* situation S *then* action A
- *if* conditions C1 and C2 hold *then* condition C does not hold

If-then rules usually turn out to be a natural form of expressing knowledge, and have the following additional desirable features:

- *Modularity*: each rule defines a small, relatively independent piece of knowledge.
- *Incrementability*: new rules can be added to the knowledge base relatively independently of other rules
- *Modifiability* (as a consequence of modularity): old rules can be changed relatively independently of other rules.

- Support system's *transparency*.

This last property is an important and distinguishing feature of expert systems. By transparency of the system we mean the system's ability to explain its decisions and solutions. If-then rules facilitate answering the following basic types of user's questions:

(1) 'How' questions: *How* did you reach this conclusion?
(2) 'Why' questions: *Why* are you interested in this information?

Mechanisms, based on if-then rules, for answering such questions will be discussed later.

If-then rules often define logical relations between concepts of the problem domain. Purely logical relations can be characterized as belonging to 'categorical knowledge', 'categorical' because they are always meant to be absolutely true. However, in some domains, such as medical diagnosis, 'soft' or probabilistic knowledge prevails. It is 'soft' in the sense that empirical regularities are usually only valid to a certain degree (often but not always). In such cases if-then rules are modified by adding a probabilistic qualification to their logical interpretation. For example:

if condition A *then* conclusion B follows *with certainty* F

Figures 14.2, 14.3 and 14.4 give an idea of the variety of ways of expressing knowledge by if-then rules. They show example rules from three different knowledge-based systems: MYCIN for medical consultation, AL/X for diagnosing equipment failures and AL3 for problem solving in chess.

In general, if you want to develop a serious expert system for some chosen domain then you have to consult actual experts for that domain and learn a great deal about it yourself. Extracting some understanding of the domain from experts and literature, and moulding this understanding into a chosen knowledge-representation formalism is called the art of *knowledge*

---

if

1 the infection is primary bacteremia, and
2 the site of the culture is one of the sterilesites, and
3 the suspected portal of entry of the organism is the gastrointestinal tract

then

there is suggestive evidence (0.7) that the identity of the organism is bacteroides.

Figure 14.2 An if-then rule from the MYCIN system for medical consultation (Shortliffe 1976). The parameter 0.7 says to what degree the rule can be trusted.

engineering. This is, as a rule, a complex effort that we cannot afford here. But we do need some domain and some knowledge base as material to experiment with. For practical reasons some toy knowledge base will have to do. Figure 14.5 shows part of such a knowledge base. It consists of simple rules that help identify animals from their basic characteristics, assuming that the identification problem is limited just to a small number of animals.

Rules in this knowledge base are of the form:

RuleName : if Condition then Conclusion.

---

if

the pressure in V-01 reached relief valve lift pressure

then

the relief valve on V-01 has lifted [N=0.005, S=400]

if

NOT the pressure in V-01 reached relief valve lift pressure, and the relief valve on V-01 has lifted

then

the V-01 relief valve opened early (the set pressure has drifted) [N=0.001, S=2000]

Figure 14.3 Two rules from an AL/X demonstration knowledge base for fault diagnosis (Reiter 1980). N and S are the 'necessity' and 'sufficiency' measures described in detail in Section 14.7. S estimates to what degree the condition part of a rule implies the conclusion part. N estimates to what degree the truth of the condition part is necessary for the conclusion to be true.

---

if

1 there is a hypothesis, H, that a plan P succeeds, and
2 there are two hypotheses,
  H_1, that a plan R_1 refutes plan P, and
  H_2, that a plan R_2 refutes plan P, and
3 there are facts: H_1 *is false*, and H_2 *is false*

then

1 generate the hypothesis, H_3, that the combined plan 'R_1 *or* R_2' refutes plan P, and
2 generate the fact: H_3 *implies not(H)*

Figure 14.4 A rule for plan refinement in chess problem solving from the AL3 system (Bratko 1982).

```
% A small knowledge base for identifying animals

:- op( 100, xfx, [has, gives, 'does not', eats, lays, isa] ).
:- op( 100, xf, [swims, flies] ).


rule1 : if
            Animal has hair
            or
            Animal gives milk
        then
            Animal isa mammal.

rule2 : if
            Animal has feathers
            or
            Animal flies and
            Animal lays eggs
        then
            Animal isa bird.

rule3 : if
            Animal isa mammal and
            ( Animal eats meat
              or
              Animal has pointed teeth and
              Animal has claws and
              Animal has 'forward pointing eyes')
        then
            Animal isa carnivore.


rule4 : if
            Animal isa carnivore and
            Animal has 'tawny colour' and
            Animal has 'dark spots'
        then
            Animal isa cheetah.


rule5 : if
            Animal isa carnivore and
            Animal has 'tawny colour' and
            Animal has 'black stripes'
        then
            Animal isa tiger.
```

```
rule6 : if
            Animal isa bird and
            Animal 'does not' fly and
            Animal swims
        then
            Animal isa penguin.

rule7 : if
            Animal isa bird and
            Animal isa 'good flyer'
        then
            Animal isa albatross.

fact : X isa animal :-
  member( X, [cheetah, tiger, penguin, albatross] ).

askable( _ gives _, 'Animal' gives 'What').
askable( _ flies, 'Animal' flies).
askable( _ lays eggs, 'Animal' lays eggs).
askable( _ eats _, 'Animal' eats 'What').
askable( _ has _, 'Animal' has 'Something').
askable( _ 'does not' _, 'Animal' 'does not' 'DoSomething').
askable( _ swims, 'Animal' swims).
askable( _ isa 'good flier', 'Animal' isa 'good flier').
```

Figure 14.5 A simple knowledge base for identifying animals. Adapted from Winston (1984). The relation 'askable' defines those things that can be asked of the user. The operators :, **if**, **then**, **and**, **or** are declared as in Figure 14.10.

where **Conclusion** is a simple assertion, and **Condition** is a set of simple assertions combined by the operators **and** and **or**. We will also allow for the operator **not** to be used in the condition part of rules, although with some reservations. By an appropriate Prolog definition of operators (as in Figure 14.5) these rules are syntactically legal Prolog clauses. The operator **and** binds stronger than **or**, which is the normal convention.

Let us consider another small knowledge base which can help locating failures in a simple electric network that consists of some electric devices and fuses. Such a network is shown in Figure 14.6. One rule can be:

if
    light1 is on *and*
    light1 is not working *and*
    fuse1 is proved intact
then
    light1 is proved broken.

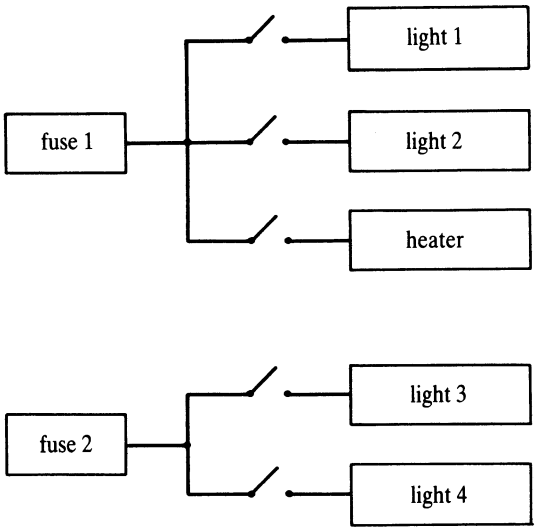


Figure 14.6 Connections between fuses and devices in a simple electric network.

Another rule can be:

if
heater is working
then
fuse1 is proved intact.

These two rules already rely on the facts (about our *particular* network) that *light1* is connected to *fuse1*, and that *light1* and *heater* share the same fuse. For another network we would need another set of rules. Therefore it is better to state rules more generally, using Prolog variables, so that they can be used for any network, and then add some extra information about a particular network. Thus one useful rule may be: if a device is on and not working and its fuse is intact then the device is broken. This translates into our rule formalism as:

broken_rule : if **Device is on and**
not (Device is working) and
Device is connected to Fuse and
Fuse is proved intact
then
Device is proved broken.

A knowledge base of this sort is shown in Figure 14.7.

```
% A small knowledge base for locating faults in an electric network

% If a device is on and not working and its fuse is intact
% then the device is broken

broken_rule :
        if
            on( Device) and
            device( Device) and
            not working( Device) and
            connected( Device, Fuse) and
            proved( intact( Fuse) )
        then
            proved( broken( Device) ).


% If a unit is working then its fuse is OK

fuse_ok_rule :
        if
            connected( Device, Fuse) and
            working( Device)
        then
            proved( intact( Fuse) ).


% If two different devices are connected to a fuse and
% are both on and not working then the fuse has failed
% NOTE: This assumes that at most one device is broken!

fused_rule :
        if
            connected( Device1, Fuse) and
            on( Device1) and
            not working( Device1) and
            samefuse( Device2, Device1) and
            on( Device2) and
            not working( Device2)
        then
            proved( failed( Fuse) ).

same_fuse_rule :
        if
            connected( Device1, Fuse) and
            connected( Device2, Fuse) and
            different( Device1, Device2)
        then
            samefuse( Device1, Device2).
```

```
fact : different( X, Y) :- not ( X = Y).

fact : device( heater).
fact : device( light1).
fact : device( light2).
fact : device( light3).
fact : device( light4).

fact : connected( light1, fuse1).
fact : connected( light2, fuse1).
fact : connected( heater, fuse1).
fact : connected( light3, fuse2).
fact : connected( light4, fuse2).

askable( on( D), on( 'Device') ).
askable( working( D), working( 'Device') ).
```

Figure 14.7 A knowledge base for locating a fault in a network such as the one in Figure 14.6.

Exercises

14.1 Consider the if-then rules of Figures 14.2 to 14.4 and translate them into our rule notation. Propose extensions to the notation to handle certainty measures when needed.

14.2 Think of some decision problem and try to formulate the corresponding knowledge in the form of if-then rules. You may consider choice of holiday, weather prediction, simple medical diagnosis and treatment, etc.

14.4 Developing the shell

Looking at the rules of our small knowledge bases of Figures 14.5 and 14.7, we can immediately see that they are, in their meaning, equivalent to Prolog's own rules. Syntactically, however, as they stand, they are merely facts from the Prolog point of view. To get these rules to work, one simple idea would be to rewrite them as actual Prolog rules. For example:

```
Animal isa mammal :-
  Animal has hair;
  Animal gives milk.

Animal isa carnivore :-
  Animal isa mammal,
  Animal eats meat.

...
```

Now a tiger called Peter can be confirmed a tiger by adding, as Prolog facts, some of Peter's properties:

```
peter has hair.
peter is lazy.
peter is big.
peter has 'tawny colour'.
peter has 'black stripes'.
peter eats meat.
```

Then we can ask:

```
?- peter isa tiger.

yes

?- peter isa cheetah.

no
```

Although Prolog does answer questions using our knowledge base, this cannot quite qualify as expert behaviour for at least two reasons:

(1) We cannot now ask for explanation; for example, *how* was it established that Peter is a tiger and *why* Peter is *not* a cheetah.

(2) All relevant information had to be input into the system (as Prolog facts) before any question was asked. This way the user may input (as in our example) some irrelevant information, or may leave out some crucial information. In the first case the user will do some unnecessary work, in the latter case the system will produce wrong answers.

To rectify these two drawbacks we need a better interaction between the user and the system during and after the reasoning process. Let us set as our goal that the system be capable of interacting with the user as the following example conversation illustrates (user's responses are in boldface and Prolog's output is in italics):

Question, please:
peter isa tiger.

Is it true: peter has hair?
yes.

Is it true: peter eats meat?
no.

Is it true: peter has pointed teeth?
yes.

Is it true: peter has claws?
why.

To investigate, by rule3, peter isa carnivore
To investigate, by rule5, peter isa tiger
This was your question

Is it true: peter has claws?
yes.

Is it true: peter has forward pointing eyes?
yes.

Is it true: peter has tawny colour?
yes.

Is it true: peter has black stripes?
yes.

(peter isa tiger) is true

Would you like to see how?
yes.

peter isa tiger
was derived by rule5 from
peter isa carnivore
was derived by rule3 from
peter isa mammal
was derived by rule1 from
peter has hair
was told
and
peter has pointed teeth
was told
and
peter has claws
was told
and
peter has forward pointing eyes
was told
and
peter has tawny colour
was told
and
peter has black stripes
was told

As this conversation shows, the system asks questions of the user about 'primitive' information, such as:

Is it true: peter eats meat?

Such information cannot be found in the knowledge base or derived from other information. The user can respond to such queries in two ways:

(1) by supplying the relevant information as an answer to the query, or
(2) ask the system *why* this information is needed.

The latter option is useful in order to enable the user to get insight into the system's current intentions. The user will ask 'why' in cases that the system's query appears irrelevant, or in cases that answering the query would require additional effort on the part of the user. From the system's explanation the user will judge whether the information the system is asking for is worth the extra effort of obtaining that information. Suppose, for example, the system is asking 'Does the animal eat meat?'. Then the user, not yet knowing the answer and not seeing the animal eating anything may decide that it is not worth waiting to actually catch the animal at eating meat.

We might use Prolog's trace facility in order to obtain some insight into the system's reasoning process. But such a trace facility would normally prove to be too rigid for our purpose. So, instead of using Prolog's own interpreting mechanism, which falls short of this type of user interaction, we will build a special interpreter facility on top of Prolog. This new interpreter will include a user-interaction facility.

14.4.1 Outline of the reasoning process

The interpreter will accept a question and find an answer. Our rule language allows for AND and OR combinations of conditions. An input question can be such a combination of subquestions. Exploring questions will therefore be similar to searching AND/OR graphs, discussed in Chapter 13.

An answer to a given question can be found in several ways, according to the following principles:

To find an answer *Answ* to a question *Q* use one of the following:

- **if *Q* is found as a fact in the knowledge base then *Answ* is '*Q* is true'**
- **if there is a rule in the knowledge base of the form**

  'if *Condition* then *Q*'

  then explore *Condition* in order to find answer *Answ*.
- **if *Q* is an 'askable' question then ask the user about *Q*.**

- if *Q* is of the form *Q1 and Q2* then explore *Q1* and now:
  if *Q1* is false then *Answ* is '*Q* is false', else explore *Q2* and appropriately combine answers to both *Q1* and *Q2* into *Answ*.
- if *Q* is of the form *Q1 or Q2* then explore *Q1* and now:
  if *Q1* is true then *Answ* is '*Q* is true', or alternatively explore *Q2* and appropriately combine answers to both *Q1* and *Q2* into *Answ*.

Questions of the form

not Q

are more problematic and will be discussed later.

14.4.2 Answering 'why' questions

A '*why*' question occurs when the system asks the user for some information and the user wants to know *why* this information is needed. Suppose that the system has asked:

Is a true?

The user may reply:

why?

An appropriate explanation can be along the following line:

Because:
I can use a to investigate b by rule R_a, and
I can use b to investigate c by rule R_b, and
I can use c to investigate d by rule R_c, and
...
I can use y to investigate z by rule R_y, and
z was your original question.

The explanation consists of showing the purpose of the information asked of the user. The purpose is shown in terms of a chain of rules and goals that connect this piece of information with the user's original question. We will call such a chain a *trace*. We can visualize a trace as a chain of rules that connects the currently explored goal and the top goal in an AND/OR tree of questions. Figure 14.8 illustrates. So, the answering of 'why' queries is accomplished by moving from the current goal upwards in the search space toward the top goal. To be able to do that we have to maintain the trace explicitly during the reasoning process.

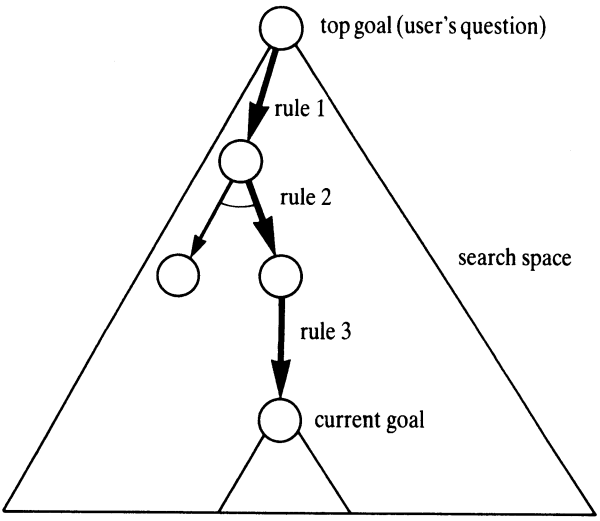


Figure 14.8 The 'why' explanation. The question 'Why are you interested in the current goal' is explained by the chain of rules and goals between the current goal and the user's original question at the top. This chain is called a trace.

14.4.3 Answering 'how' questions

Once the system has come up with an answer to the user's question, the user may like to see *how* this conclusion was reached. A proper way of answering such a 'how' question is to display the evidence: that is, rules and subgoals from which the conclusion was reached. For our rule language, such evidence consists of an AND/OR solution tree. Therefore our inference engine will have to produce as an answer an AND/OR solution tree comprised of rule names and subgoals. Just answering the top goal would not be sufficient. Such a tree can then be displayed as a 'how' explanation. A suitable display form can be achieved by properly indenting subtrees. For example:

```
peter isa carnivore
  was derived by rule3 from
    peter isa mammal
      was derived by rule1 from
        peter has hair
          was told
    and
    peter eats meat
      was told
```

14.5 Implementation

We will now implement our shell along the ideas developed in the previous section. Figure 14.9 illustrates the main objects manipulated by the shell. **Goal** is a question to be investigated; **Trace** is a chain of ancestor goals and rules between **Goal** and the top-level question; **Answer** is an AND/OR-type solution tree for **Goal**.

The main procedures of the shell will be:

explore(Goal, Trace, Answer)

which finds an answer **Answer** to a question **Goal**;

useranswer(Goal, Trace, Answer)

generates solutions for an 'askable' **Goal** by asking the user about **Goal** and answers 'why' questions;

present(Answer)

displays the result and answers 'how' questions. These procedures are properly put into execution by the 'driver' procedure **expert**.

14.5.1 Procedure explore

The heart of the shell is the procedure

explore(Goal, Trace, Answer)

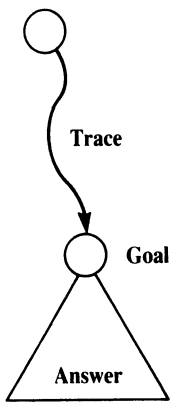


Figure 14.9 The relation **explore(Goal, Trace, Answer)**. **Answer** is an AND/OR solution tree for **Goal**.

which will find an answer **Answer** to a given question **Goal** by using the principles outlined in Section 14.4.1: either find **Goal** as a fact in the knowledge base, or apply a rule in the knowledge base, or ask the user, or treat **Goal** as an AND or OR combination of subgoals.

The meaning and the structure of the arguments are as follows:

Goal is a question to be investigated, represented as an AND/OR combination of simple assertions. For example:

X has feathers or X flies and X lays eggs

Trace is a chain of ancestor goals and rules between **Goal** and the original, top goal, represented as a list of items of the form

Goal by Rule

This means that **Goal** is being investigated by means of rule **Rule**. For example, let the top goal be 'peter isa tiger', and the currently investigated goal be 'peter eats meat'. The corresponding trace, according to the knowledge base of Figure 14.5, is:

[(peter isa carnivore) by rule3, (peter isa tiger) by rule5]

This intends to say the following:

I can use 'peter eats meat' in order
to investigate, by rule3, 'peter isa carnivore'.
Further, I can use 'peter isa carnivore' in order
to investigate, by rule5, 'peter isa tiger'.

Answer is an AND/OR solution tree for the question **Goal**. The general form for **Answer** is:

Conclusion was Found

where **Found** represents a justification for **Conclusion**. The following three example answers illustrate different possibilities:

(1) **(connected(heater, fuse1)) was 'found as a fact'**

(2) **(peter eats meat) was told**

(3) **(peter isa carnivore) was ('derived by' rule3 from**
(peter isa mammal) was ('derived by' rule1 from
(peter has hair) was told) and
(peter eats meat) was told)

Figure 14.10 shows the Prolog code for **explore**. This code implements the principles of Section 14.4.1, using the data structures specified above.

```
% Procedure
%
% explore( Goal, Trace, Answer)
%
% finds Answer to a given Goal. Trace is a chain of ancestor
% goals and rules. 'explore' tends to find a positive answer
% to a question. Answer is 'false' only when all the
% possibilities have been investigated and they all resulted
% in 'false'.

:- op( 900, xfx, :).
:- op( 800, xfx, was).
:- op( 870, fx, if).
:- op( 880, xfx, then).
:- op( 550, xfy, or).
:- op( 540, xfy, and).
:- op( 300, fx, 'derived by').
:- op( 600, xfx, from).
:- op( 600, xfx, by).


explore( Goal, Trace, Goal is true was 'found as a fact') :-
  fact : Goal.

% Assume only one rule about each type of goal

explore( Goal, Trace,
         Goal is TruthValue was 'derived by' Rule from Answer) :-
  Rule : if Condition then Goal,              % Rule relevant to Goal
  explore( Condition, [Goal by Rule | Trace], Answer),
  truth( Answer, TruthValue).

explore( Goal1 and Goal2, Trace, Answer) :- !,
  explore( Goal1, Trace, Answer1),
  continue( Answer1, Goal1 and Goal2, Trace, Answer).

explore( Goal1 or Goal2, Trace, Answer) :-
  exploreyes( Goal1, Trace, Answer);          % Positive answer to Goal1
  exploreyes( Goal2, Trace, Answer).          % Positive answer to Goal2

explore( Goal1 or Goal2, Trace, Answer1 and Answer2) :- !,
  not exploreyes( Goal1, Trace, _),
  not exploreyes( Goal2, Trace, _),           % No positive answer
  explore( Goal1, Trace, Answer1),            % Answer1 must be negative
  explore( Goal2, Trace, Answer2).            % Answer2 must be negative

explore( Goal, Trace, Goal is Answer was told) :-
  useranswer( Goal, Trace, Answer).           % User-supplied answer
```

```
exploreyes( Goal, Trace, Answer) :-
  explore( Goal, Trace, Answer),
  positive( Answer).

continue( Answer1, Goal1 and Goal2, Trace, Answer) :-
  positive( Answer1),
  explore( Goal2, Trace, Answer2),
  ( positive( Answer2), Answer = Answer1 and Answer2;
    negative( Answer2), Answer = Answer2).

continue( Answer1, Goal1 and Goal2, _, Answer1) :-
  negative( Answer1).

truth( Question is TruthValue was Found, TruthValue) :- !.

truth( Answer1 and Answer2, TruthValue) :-
  truth( Answer1, true),
  truth( Answer2, true), !,
  TruthValue = true;
  TruthValue = false.

positive( Answer) :-
  truth( Answer, true).

negative( Answer) :-
  truth( Answer, false).
```

Figure 14.10 The core procedure of an expert system shell.

14.5.2 Procedure *useranswer*

Before developing **useranswer** let us consider a useful auxiliary procedure

getreply(Reply)

How about "I don't know"?

During conversation, the user is often expected to reply with 'yes', 'no' or 'why'. The purpose of **getreply** is to extract such an answer from the user and to also understand it properly if the user abbreviates ('y' or 'n') or makes a typing error. If the user's reply is unintelligible then **getreply** will request another reply from the user.

```
getreply( Reply) :-
  read( Answer),
  means( Answer, Meaning), !,          % Answer means something?
  Reply = Meaning;                     % Yes
  nl, write( 'Answer unknown, try again please'), nl,  % No
  getreply( Reply).                    % Try again
```

```
means( yes, yes).
means( y, yes).
means( no, no).
means( n, no).
means( why, why).
means( w, why).
```

Note that **getreply** should be used with care because it involves interaction with the user. For example, the following is a bad attempt to interpret the user's reply:

```
getreply( yes), interpretyes( ...);
getreply( no), interpretno( ...);
...
```

For example, if the user types 'no' the program will want this answer repeated. Therefore a better way is:

```
getreply( Reply),
( Reply = yes, interpretyes( ...);
  Reply = no, interpretno( ...);
  ... )
```

The procedure

```
useranswer( Goal, Trace, Answer)
```

asks the user about **Goal**. **Answer** is the result of this inquiry. **Trace** is used for explanation in the case that the user asks 'why'.

useranswer should first check whether **Goal** is a kind of information that can be asked of the user. In our shell, such kinds of goal will be called 'askable', defined for now by a relation

```
askable( Goal)
```

This will be refined later. If **Goal** is 'askable' then **Goal** is displayed and the user will specify whether it is true or false. In the case that the user asks 'why', **Trace** will be displayed. If **Goal** is true then the user will also specify the values of variables in **Goal** (if there are any).

This can be programmed as a first attempt as follows:

```
useranswer( Goal, Trace, Answer) :-
  askable( Goal),                    % Can Goal be asked of the user?
  ask( Goal, Trace, Answer).         % Ask user about Goal
```

```
ask( Goal, Trace, Answer) :-
  introduce( Goal),                          % Show question to user
  getreply( Reply),                          % Read user's reply
  process( Reply, Goal, Trace, Answer).      % Process the reply

process( why, Goal, Trace, Answer) :-        % User is asking 'why'
  showtrace( Trace),                         % Show why
  ask( Goal, Trace, Answer).                 % Ask again

process( yes, Goal, Trace, Answer) :-        % User says Goal is true
  Answer = true,
  askvars( Goal);                            % Ask about variables
  ask( Goal, Trace, Answer).                 % Ask for more solutions

process( no, Goal, Trace, false).            % User says Goal is false

introduce( Goal) :-
  nl, write( 'Is it true: '),
  write( Goal), write( ?), nl.
```

The call **askvars(Goal)** will ask the user to specify the value of each variable in **Goal**:

```
askvars( Term) :-
  var( Term), !,                             % A variable?
  nl, write( Term), write( ' = '),
  read( Term).                               % Read variable's value

askvars( Term) :-
  Term =.. [Functor | Args],                 % Get arguments of a structure
  askarglist( Args).                         % Ask about variables in arguments

askarglist( [] ).

askarglist( [Term | Terms] ) :-
  askvars( Term),
  askarglist( Terms).
```

Let us make a few experiments with this **useranswer** procedure. For example, let the binary relation **eats** be declared as 'askable':

```
askable( X eats Y).
```

(In the following dialogue between Prolog and the user, the user-typed text is in boldface and Prolog's output is in italics.)

```
?- useranswer( peter eats meat, [], Answer).

Is it true: peter eats meat?                 % Question to user
yes.                                         % User's reply

Answer = true
```

A more interesting example that involves variables may look like this:

```
?- useranswer( Who eats What, [], Answer).

Is it true: _17 eats _18?          % Prolog gives internal names to variables
yes.
_17 = peter.                       % Asking about variables
_18 = meat.

Answer = true
Who = peter
What = meat;                       % Backtrack for more solutions

Is it true: _17 eats _18?
yes.
_17 = susan.
_18 = bananas.

Answer = yes
Who = susan
What = bananas;

Is it true: _17 eats _18?
no.

Answer = no
```

14.5.3 Refining *useranswer*

One drawback of our **useranswer** procedure that shows in the foregoing conversation is the awkward appearance of Prolog-generated variable names in Prolog's output. Symbols like _17 should be replaced by some meaningful words when displayed to the user.

Another, more serious, defect of this version of **useranswer** is the following. If we subsequently use **useranswer** on the same goal, the user will have to repeat all the solutions. If our expert system would, during its reasoning process, come to explore the same 'askable' goal twice it would bore the user with exactly the same conversation again, instead of using the information previously supplied by the user.

Let us now rectify these two defects. First, an improvement of the external appearance of queries to the user can be based on introducing some standard format for each 'askable' goal. To this end, the relation **askable** can be added a second argument which will specify this format, as shown by the following example:

```
askable( X eats Y, 'Animal' eats 'Something').
```

In querying the user, each variable in the question should then be replaced by keywords in the question format. For example:

```
?- useranswer( X eats Y, [], Answer).

Is it true: Animal eats Something?
yes.
Animal = peter.
Something = meat.

Answer = true
X = peter
Y = meat
```

In an improved version of **useranswer**, shown in Figure 14.11, this formatting of queries is done by the procedure:

format(Goal, ExternFormat, Question, Vars0, Variables)

```
% Procedure
%
% useranswer( Goal, Trace, Answer)
%
% generates, through backtracking, user-supplied solutions to Goal.
% Trace is a chain of ancestor goals and rules used for 'why'
% explanation.

useranswer( Goal, Trace, Answer) :-
  askable( Goal, _),                    % May be asked of the user
  freshcopy( Goal, Copy),               % Variables in Goal renamed
  useranswer( Goal, Copy, Trace, Answer, 1).

% Do not ask again about an instantiated goal

useranswer( Goal, _, _, _, N) :-
  N > 1,                                % Repeated question?
  instantiated( Goal), !,
  fail.                                 % Do not ask again

% Is Goal implied true or false for all instantiations?

useranswer( Goal, Copy, _, Answer, _) :-
  wastold( Copy, Answer, _),
  instance_of( Copy, Goal), !.          % Answer to Goal implied

% Retrieve known solutions, indexed from N on, for Goal

useranswer( Goal, _, _, true, N) :-
  wastold( Goal, true, M),
  M >= N.
```

```
% Has everything already been said about Goal?

useranswer( Goal, Copy, _, Answer, _) :-
  end_answers( Copy),
  instance_of( Copy, Goal), !,          % Everything was already said about Goal
  fail.

% Ask the user for (more) solutions

useranswer( Goal, _, Trace, Answer, N) :-
  askuser( Goal, Trace, Answer, N).

askuser( Goal, Trace, Answer, N) :-
  askable( Goal, ExternFormat),
  format( Goal, ExternFormat, Question, [], Variables), % Get question format
  ask( Goal, Question, Variables, Trace, Answer, N).

ask( Goal, Question, Variables, Trace, Answer, N) :-
  nl,
  ( Variables = [], !,                          % Introduce question
    write( 'Is it true: ');
    write( 'Any (more) solution to: ') ),
  write( Question), write( '? '),
  getreply( Reply), !,                          % Reply = yes/no/why
  process( Reply, Goal, Question, Variables, Trace, Answer, N).

process( why, Goal, Question, Variables, Trace, Answer, N) :-
  showtrace( Trace),
  ask( Goal, Question, Variables, Trace, Answer, N).

process( yes, Goal, _, Variables, Trace, true, N) :-
  nextindex( Next),                     % Get new free index for 'wastold'
  Next1 is Next + 1,
  ( askvars( Variables),
    assertz( wastold( Goal, true, Next) );          % Record solution
    freshcopy( Goal, Copy),                         % Copy of Goal
    useranswer( Goal, Copy, Trace, Answer, Next1) ). % More answers?

process( no, Goal, _, _, _, false, N) :-
  freshcopy( Goal, Copy),
    wastold( Copy, true, _), !,         % 'no' means: no more solutions
    assertz( end_answers( Goal) ),      % Mark end of answers
    fail;
  nextindex( Next),                     % Next free index for 'wastold'
    assertz( wastold( Goal, false, Next) ).     % 'no' means: no solution

format( Var, Name, Name, Vars, [Var/Name | Vars] ) :-
  var( Var), !.

format( Atom, Name, Atom, Vars, Vars) :-
  atomic( Atom), !,
  atomic( Name).
```

```
format( Goal, Form, Question, Vars0, Vars) :-
  Goal =.. [Functor | Args1],
  Form =.. [Functor | Forms],
  formatall( Args1, Forms, Args2, Vars0, Vars),
  Question =.. [Functor | Args2].

formatall( [], [], [], Vars, Vars).

formatall( [X | XL], [F | FL], [Q | QL], Vars0, Vars) :-
  formatall( XL, FL, QL, Vars0, Vars1),
  format( X, F, Q, Vars1, Vars).

askvars( [] ).

askvars( [Variable/Name | Variables] ) :-
  nl, write( Name), write( ' = '),
  read( Variable),
  askvars( Variables).

showtrace( [] ) :-
  nl, write( 'This was your question'), nl.

showtrace( [Goal by Rule | Trace] ) :-
  nl, write( 'To investigate, by '),
  write( Rule), write( ', '),
  write( Goal),
  showtrace( Trace).


% instance-of( T1, T2) means instance of T1 is T2; that is,
% term T1 is more general than T2 or equally general as T2

instance_of( Term, Term1) :-      % Instance of Term is Term1
  freshcopy( Term1, Term2),       % Copy of Term1 with fresh set of variables
  numbervars( Term2, 0, _), !,
  Term = Term2.                   % This succeeds if Term1 is instance of Term

freshcopy( Term, FreshTerm) :-    % Make a copy of Term with variables renamed
  asserta( copy( Term) ),
  retract( copy( FreshTerm) ), !.

lastindex( 0).                    % Index for 'wastold' at start

nextindex( Next) :-               % Next free index for 'wastold'
  retract( lastindex( Last) ), !,
  Next is Last + 1,
  assert( lastindex( Next) ).
```

Figure 14.11 Expert system shell: Querying the user and answering 'why' questions.

Goal is a goal to be formatted. **ExternFormat** specifies the external format for **Goal**, defined by:

```
askable( Goal, ExternFormat)
```

Question is **Goal** formatted according to **ExternFormat**. **Variables** is a list of variables that appear in **Goal** accompanied by their corresponding keywords (as specified in **ExternFormat**), added on a list **Vars0**. For example:

```
?- format( X gives documents to Y,
           'Who' gives 'What' to 'Whom',
           Question, [], Variables).

Question = 'Who' gives documents to 'Whom',
Variables = [X/'Who', Y/'Whom'].
```

The other refinement, to avoid repeated questions to the user, will be more difficult. First, all user's answers should be remembered so that they can be retrieved at some later point. This can be accomplished by asserting user's answers as elements of a relation. For example:

```
assert( wastold( mary gives documents to friends, true) ).
```

In a situation where there are several user-supplied solutions to the same goal there will be several facts asserted about that goal. Here a complication arises. Suppose that variants of a goal (the goal with variables renamed) appear at several places. For example:

```
(X has Y) and                    % First occurrence - Goal1
...
(X1 has Y1) and                  % Second occurrence - Goal2
...
```

Further suppose that the user will be asked (through backtracking) for several solutions to **Goal1**. After that the reasoning process will advance to **Goal2**. As we already have some solutions for **Goal1** we want the system to apply them automatically to **Goal2** as well (since they obviously satisfy **Goal2**). Now suppose that the system tries these solutions for **Goal2**, but none of them satisfies some further goal. So the system will backtrack to **Goal2** and should ask the user for more solutions. If the user does supply more solutions then these will have to be remembered as well. In the case that the system later backtracks to **Goal1** these new solutions will also have to be automatically applied to **Goal1**.

In order to properly use the information supplied by the user at different places we will index this information. So the asserted facts will have the form

```
wastold( Goal, TruthValue, Index)
```

where **Index** is a counter of user-supplied answers. The procedure

useranswer(Goal, Trace, Answer)

will have to keep track of the number of solutions already produced through backtracking. This can be accomplished by means of another procedure, **useranswer** with four arguments,

useranswer(Goal, Trace, Answer, N)

where N is an integer. This call has to produce solutions to **Goal** indexed N or higher. A call

useranswer(Goal, Trace, Answer)

is meant to produce *all* solutions to **Goal**. Solutions will be indexed from 1 on, so we have the following relation:

```
useranswer( Goal, Trace, Answer) :-
  useranswer( Goal, Trace, Answer, 1).
```

An outline of

useranswer(Goal, Trace, Answer, N)

is: generate solutions to **Goal** by first retrieving known solutions indexed from N onwards. When these are exhausted then start querying the user about **Goal** and assert the thus obtained new solutions properly indexed by consecutive numbers. When the user says there are no more solutions, assert

end_answers(Goal)

If the user says in the first place that there are no solutions at all then assert

wastold(Goal, false, Index)

When retrieving solutions, **useranswer** will have to properly interpret such information.

However, there is a further complication. The user may also specify general solutions, leaving some variables uninstantiated. If a positive solution is retrieved which is more general than or as general as **Goal**, then there is of course no point in further asking about **Goal** since we already have the most general solution. If

wastold(Goal, false, _)

then an analogous decision is to be made.

The **useranswer** program in Figure 14.11 takes all this into account. Another argument, **Copy** (a copy of **Goal**), is added and used in several matchings in place of **Goal** so that the variables of **Goal** are not destroyed. The program also uses two auxiliary relations. One is

instantiated(Term)

which is true if **Term** contains no variables. The other is

instance_of(Term, Term1)

where **Term1** is an instance of **Term**; that is, **Term** is at least as general as **Term1**. For example:

instance_of(X gives information to Y, mary gives information to Z)

These two procedures both rely on another procedure:

numbervars(Term, N, M)

This procedure 'numbers' the variables in **Term** by replacing each variable in **Term** by some newly generated term so that these 'numbering' terms correspond to integers between N and M – 1. For example, let these terms be of the form

var/0, var/1, var/2, ...

Then

?- Term = f(X, t(a, Y, X)), numbervars(Term, 5, M).

will result in:

```
Term = f( var/5, t( a, var/6, var/5) )
M = 7
```

Such a **numbervars** procedure is often supplied as a built-in predicate in a Prolog system. If not, it can be programmed as follows:

```
numbervars( Term, N, Nplus1) :-
  var( Term), !,                    % Variable?
  Term = var/N,
  Nplus1 is N + 1.

numbervars( Term, N, M) :-
  Term =.. [Functor | Args],        % Structure or atomic
  numberargs( Args, N, M).          % Number variables in arguments
```

```
% Displaying the conclusion of a consultation and 'how' explanation

present( Answer) :-
  nl, showconclusion( Answer),
  nl, write( 'Would you like to see how? '),
  getreply( Reply),
  ( Reply = yes, !, show( Answer);          % Show solution tree
    true ).

showconclusion( Answer1 and Answer2) :- !,
  showconclusion( Answer1), write( ' and '),
  showconclusion( Answer2).

showconclusion( Conclusion was Found) :-
  write( Conclusion).


% 'show' displays a complete solution tree

show( Solution) :-
  nl, show( Solution, 0), !.                % Indent by 0

show( Answer1 and Answer2, H) :- !,         % Indent by H
  show( Answer1, H),
  tab( H), write( and), nl,
  show( Answer2, H).

show( Answer was Found, H) :-               % Indent by H
  tab( H), writeans( Answer),               % Show conclusion
  nl, tab( H),
  write( ' was '),
  show1( Found, H).                         % Show evidence

show1( Derived from Answer, H) :- !,
  write( Derived), write( ' from'),         % Show rule name
  nl, H1 is H + 4,
  show( Answer, H1).                        % Show antecedent

show1( Found, _) :-                         % Found = 'told' or 'found as fact'
  write( Found), nl.

writeans( Goal is true) :- !,
  write( Goal).                             % Omit 'is true' on output

writeans( Answer) :-                        % This is negative answer
  write( Answer).
```

Figure 14.12 Expert system shell: Displaying final result and 'how' explanation.

```
numberargs( [], N, N) :- !.

numberargs( [X | L], N, M) :-
  numbervars( X, N, N1),
  numberargs( L, N1, M).
```

14.5.4 Procedure *present*

The procedure

```
present( Answer)
```

in Figure 14.12 displays the final result of a consultation session and generates the 'how' explanation. **Answer** includes both an answer to the user's question, and a derivation tree showing *how* this conclusion was reached. Procedure **present** first presents the conclusion. If the user then wants to see *how* the conclusion was reached, then the derivation tree is displayed in a suitable form which constitutes a 'how' explanation. This form was illustrated by an example in Section 14.4.3.

14.5.5 Top-level driver

Finally, for a handy access to the shell from the Prolog interpreter we need a 'driver' procedure which may look like the procedure **expert** in Figure 14.13. **expert** starts the execution and coordinates the three main modules of the shell shown in Figures 14.10 to 14.12. For example:

```
?- expert.

Question, please:                    % Prompt the user
X isa animal and goliath isa X.      % User's question

Is it true: goliath has hair?
...
```

14.5.6 A comment on the shell program

Our shell program at some places appears to lack the declarative clarity that is typical of Prolog programs. The reason for this is that in such a shell we have to impose a rather strict control over the execution because an expert system is expected not only to find an answer, but also to find it in a way that appears sensible to the user who keeps interacting with the system. Therefore, we have to implement a particular problem-solving *behaviour* and not just an input–output relation. Thus a resulting program is in fact more procedurally biased than usual. This is one example when we cannot rely on Prolog's own procedural engine, but have to specify the procedural behaviour in detail.

```
% Top-level driving procedure

expert :-
  getquestion( Question),                % Input user's question
  ( answeryes( Question);                % Try to find positive answer
    answerno( Question) ).               % If no positive answer then find negative

answeryes( Question) :-                  % Look for positive answers to Question
  markstatus( negative),                 % No positive answer yet
  explore( Question, [], Answer),        % Trace is empty
  positive( Answer),                     % Look for positive answers
  markstatus( positive),                 % Positive answer found
  present( Answer), nl,
  write( 'More solutions? '),
  getreply( Reply),                      % Read user's reply
  Reply = no.                            % Otherwise backtrack to 'explore'

answerno( Question) :-                   % Look for negative answer to question
  retract( no_positive_answer_yet), !,   % Has there been no positive answer?
  explore( Question, [], Answer),
  negative( Answer),
  present( Answer), nl,
  write( 'More negative solutions? '),
  getreply( Reply),
  Reply = no.                            % Otherwise backtrack to 'explore'

markstatus( negative) :-
  assert( no_positive_answer_yet).

markstatus( positive) :-
  retract( no_positive_answer_yet), !; true.

getquestion( Question) :-
  nl, write( 'Question, please'), nl,
  read( Question).
```

Figure 14.13 Expert system shell: a 'driver'. The shell is called from Prolog through the procedure **expert**.

14.5.7 Negated goals

It seems natural to allow for negation in the left-hand sides of rules and hence also in questions that are investigated by **explore**. A straightforward attempt to deal with negated questions is as follows:

```
explore( not Goal, Trace, Answer) :- !,
  explore( Goal, Trace, Answer1),
  invert( Answer1, Answer).              % Invert truth value
```

invert(Goal is true was Found, (not Goal) is false was Found).

invert(Goal is false was Found, (not Goal) is true was Found).

This is fine if **Goal** is instantiated. If it is not, problems arise. Consider, for example:

?- expert.

Question, please:
not (X eats meat).

Is it true: Somebody eats meat?
yes.
Somebody = **tiger.**

Now the system will come up with the answer:

not (tiger eats meat) is false

This is not satisfying. The problem stems from what we mean by a question such as:

not (X eats meat)

We in fact want to ask: Is there an X such that X does not eat meat? But the way this question is interpreted by **explore** (as defined) is as follows:

(1) Is there an X such that X eats meat?
(2) Yes, tiger eats meat.

Thus

(3) not (tiger eats meat) is false.

In short, the interpretation is: Is it true that no X eats meat? So we will get a positive answer only in the case that ***nobody*** eats meat. Said another way, **explore** will answer the question as if X was *universally* quantified:

for *all* X: not (X eats meat)?

and not as if it was existentially quantified, which was our intention:

for *some* X: not (X eats meat)?

If the question explored is instantiated then this problem disappears.

Otherwise, proper treatment is more complicated. Some decisions can be as follows:

To explore *not Goal*, explore *Goal* and now:

- if *Goal* is false then (*not Goal*) is true;
- if *Goal'* is a solution of *Goal*
  and Goal' is as general as *Goal*
  then (*not Goal*) is false;
- if *Goal'* is a solution of *Goal* and
  Goal' is more specific than *Goal*
  then we cannot say anything definite about *Goal*.

We can avoid these complications by only allowing instantiated negated goals. This can often be achieved by proper statement of rules in the knowledge base. In Figure 14.7 we achieved this in 'broken_rule':

```
broken_rule : if
                  on( Device) and
                  device( Device) and          % Instantiate Device
                  not working( Device) and
                  connected( Device, Fuse) and
                  intact( Fuse)
              then
                  broken( Device).
```

The condition

```
device( Device)
```

will 'protect' the subsequent condition

```
not working( Device)
```

from being evaluated uninstantiated.

Exercise

14.3 A knowledge base can in principle contain cycles. For example:

```
rule1: if bottle_empty then john_drunk.
rule2: if john_drunk then bottle_empty.
```

Using such a knowledge base, our **explore** procedure may start cycling between same goals. Modify **explore** to prevent such cycling. **Trace** can be used for this. However, some care is necessary: if the current goal *matches* a previous goal, this should not be considered a cycle if the current goal is more general than the previous one.

14.6 Dealing with uncertainty

14.6.1 Certainty measures

Our expert system shell of the previous section only deals with questions that are either true or false. Such domains in which all answers reduce to true or false are called *categorical*. As data, rules were also categorical: 'categorical implications'. However, many expert domains are not categorical. Typical expert behaviour is full of guesses (highly articulated, though) that are usually true, but there can be exceptions. Both data about a particular problem and implications in general rules can be less than certain. We can model uncertainty by assigning to assertions some qualification other than just *true* and *false*. Such qualification can be expressed by descriptors – for example, *true*, *highly likely*, *likely*, *unlikely*, *impossible*. Alternatively, the degree of belief can be expressed by a real number in some interval – for example, between 0 and 1 or –5 and +5. Such numbers are called various names, such as 'certainty factor', 'measure of belief' or 'subjective certainty'. It would be natural to use actual probabilities, but then practical problems arise for the following reasons:

- Human experts seem to have troubles thinking in terms of actual probabilities; their likelihood estimates do not quite correspond to probabilities as defined mathematically.
- Mathematically correct probabilistic treatment would require either information that is not available or some simplification assumptions that are not really quite justified in a practical application.

Therefore even if the likelihood measure is in the interval between 0 and 1, it is more appropriate to cautiously call it a 'subjective certainty' to indicate that it is an expert's estimate. Expert's estimates do not in general satisfy the requirements of the probability theory. Also, the computations with such likelihood measures may differ from the probability calculus. But nevertheless they may quite adequately model the human certainty estimates.

Many schemes for dealing with uncertainty have been developed. We will consider here one that is used in the systems Prospector and AL/X, which has been applied to mineral exporation and fault diagnosis respectively. It should be noted that the Prospector model is not perfect neither from the theoretical nor from the practical point of view. However, it has been used practically, it is simple and illustrative of the main principles, and therefore suitable at least as a first exercise. On the other hand, even much more complicated schemes are known to be not without difficulties.

14.6.2 The Prospector model

The likelihood of events is modelled by real numbers between 0 and 1. For simplicity we will be referring to these numbers as 'probabilities', although 'subjective certainties' would be more accurate. Relations between events can

be represented diagrammatically as an 'inference network'. Figure 14.14 shows an example. Boxes represent events and arcs represent relations between **events**. Circles represent logical combinations of events (AND, OR, NOT).

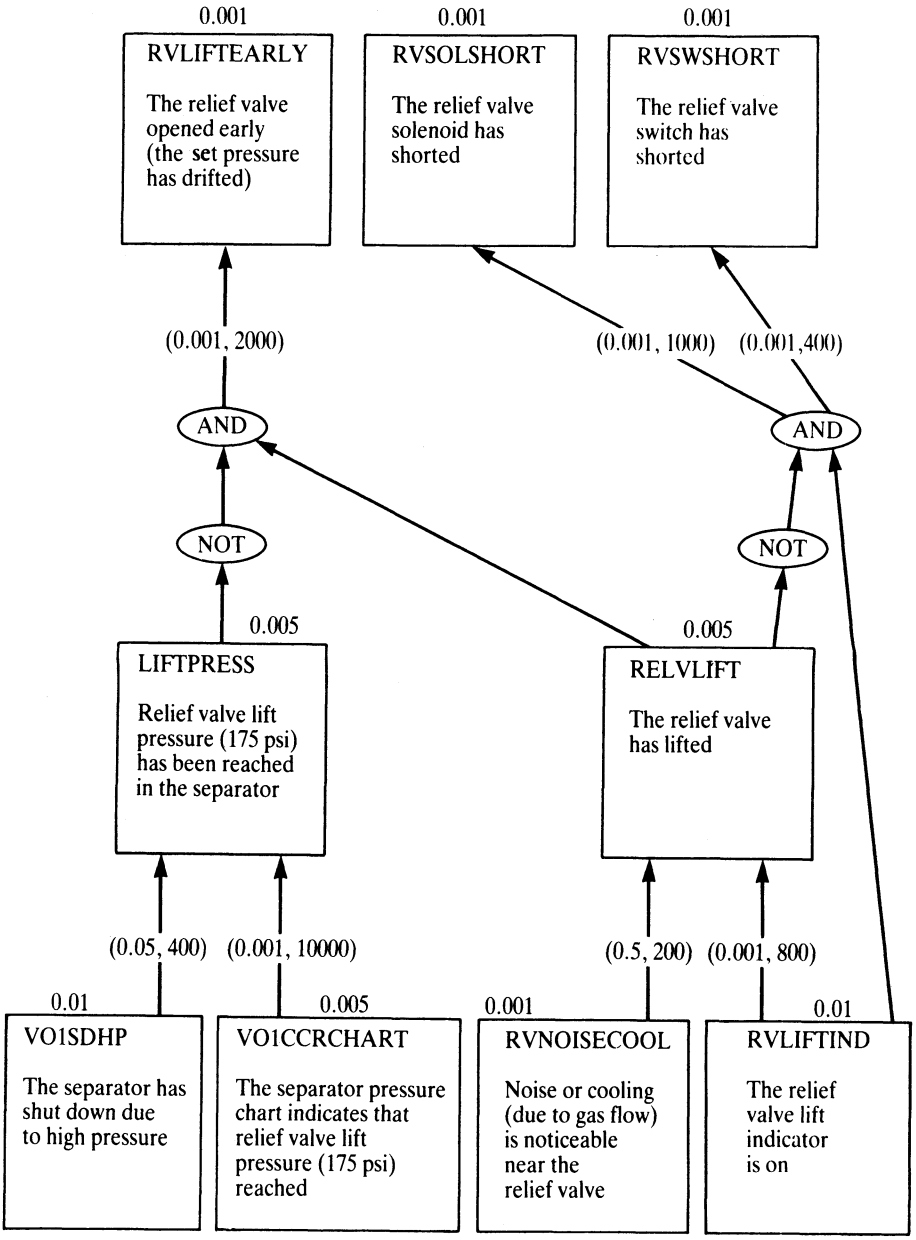


Figure 14.14 An AL/X inference network adapted from Reiter (1980). Numbers attached to boxes are prior probabilities of events; numbers attached to arcs indicate the strength of relations between events.

Relations between events, pictured as arcs, mean a sort of 'soft implication'. Let there be two events, E and H, and let the knowledge of E affect our belief in H. If this effect was that of 'categorical implication' then we would simply write:

if E then H

In the case of 'soft implication' this relation can be less than certain, so some 'strength' can be associated with it:

if E then H with strength S

The strength with which the likelihood of E influences the belief in H is in Prospector modelled by two parameters:

N = 'necessity factor'
S = 'sufficiency factor'

In an inference network, this is pictured as:

E ----------> H
(N,S)

The two events in such a relation are often called 'evidence' and 'hypothesis' respectively. Suppose that we investigate a hypothesis H. Then we collect evidence E in order to confirm or disconfirm the hypothesis. S tells how *sufficient* E is for H; N tells how *necessary* E is for H. If E is true then the greater S is the more likely H is. On the other hand, if E is false then the lower N is the less likely H is. In a case that the likelihood of E is somewhere between certain and impossible, the likelihood of H is determined by interpolation between the extreme cases. The extreme cases are:

(1) E is known to be false
(2) E is known to be true
(3) nothing is known about E

There is a prior probability $p_0(H)$ for each event H in the inference network. So $p_0(H)$ is the (unconditional) probability of event H if no positive or negative evidence is known. If there is some evidence E known then the probability of H is changed from $p_0(H)$ into $p(H|E)$. The degree of change depends on the strength of the arc between E and H in the network. So, in general, we start investigating hypotheses assuming their prior probabilities. During investigation more information is accumulated, and this will reflect in the changes of the probabilities of events in the network. These changes will propagate through the network from event to event according to the links

between events. In Figure 14.14, for example, assume that the relief-valve lift-indicator is learned to be on. This information will affect our belief that the relief valve has lifted, and this may in turn affect our belief that the set pressure has drifted.

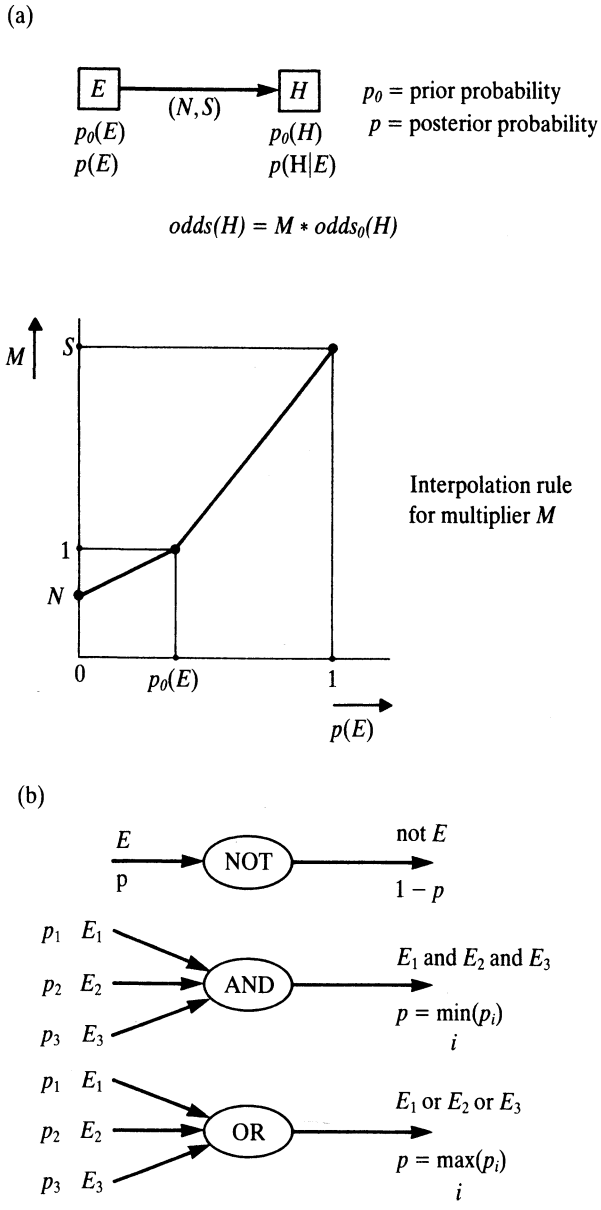


Figure 14.15 Probability propagation rules in Prospector and AL/X: (a) 'soft implication' with strength (N,S); (b) logical relations.

Figure 14.15 shows one way of implementing such propagation effects. Part of the calculation is done with *odds* instead of probabilities. This is not necessary in principle, but convenient. There is a simple relation between odds and probabilities:

$$odds = prob/(1 - prob)$$

$$prob = odds/(1 + odds)$$

Let E 'softly imply' H; then, according to Figure 14.15,

$$odds(H|E) = M * odds_0(H)$$

where the multiplier M is determined by the prior and posterior probabilities of E, and by the strength (N,S) of the link between E and H. Prospector's rules (in Figure 14.15) that combine probabilities of logically combined events (*min* and *max*) are supposed appropriately to model the human judgement regarding the combinations of subjective certaintities.

14.6.3 Outline of an implementation

Let us first extend the rule language in order to handle uncertainty. Each rule can be added a 'strength modifier' defined by two non-negative real numbers S and N. A suitable format is:

```
RuleName : if
                Condition
           then
                Conclusion
           with
                strength( N, S ).
```

Example rules from Figure 14.14 can be expressed in this form as follows:

```
rule1 : if
              not liftpress and
              relvlift
        then
              rvliftearly
        with
              strength( 0.001, 2000).

rule4 : if
              v01sdhp
        then
              liftpress
        with
              strength( 0.05, 400).
```

To extend our expert system shell of Section 14.5 to deal with uncertainty, changes are needed in most procedures. Let us concentrate just on the procedure

explore(Goal, Trace, Answer)

We will assume here that **Goal** contains no variables (as in the cases of Prospector and AL/X). This greatly simplifies matters (especially in procedure **useranswer**). So **Goal** can be a logical combination of simple propositions. For example:

not liftpress and relvlift

The chain of antecedent goals and rules, **Trace**, can be represented in the same way as in Section 14.5. The form of **Answer**, however, needs modification in order to introduce probabilities. We can combine a goal and its probability into a term of the form:

Goal : Probability

So an example of **Answer** is:

rvliftind : 1 was told

This means that the user has told the system that the event **rvliftind** certainly happened.

Another modification in the representation of **Answer** is necessary because several independent links can bear on the same event. Then the probability of this event can be multiplicatively affected by the odds multipliers, as in Figure 14.15, from all the links. In such a case, **Answer** will contain the list of all relevant derivation branches. An example of such an answer from the network of Figure 14.14 can be (properly arranged for better readability):

```
liftpress : 1 was 'derived by'
  [ rule4 from v01sdhp : 1 was told,
    rule5 from v01ccrchart : 1 was told ]
```

Procedure **explore**, which produces answers in this form, is programmed in Figure 14.16. **explore** calls the predicate

implies(P0, P, Strength, Prob0, Prob)

which is the 'soft implication' relation (see Figure 14.15). P0 is the prior

probability of the evidence, E, and P is its posterior probability. **Strength** is the strength of the implication, represented as

strength(N, S)

Prob0 and **Prob** are the prior and the posterior probability respectively of the hypothesis, H.

It should be noted that this is a very simple implementation that only caters for the probability propagation in an inference network, and may behave unintelligently. It pays no attention to what lines of analysis are currently most important. In a more elaborate version, the exploration process should be guided toward the most critical pieces of evidence. It should also aim at asking the user as few questions as possible.

Finally, a few comments about a new **useranswer** procedure. It would be simpler than that in Figure 14.11 since now we have no variables in queries to the user. This time the user will answer by a probability (instead of 'yes' or 'no'). In the case that the user knows nothing about the event asked, he or she will leave the prior probability of that event unchanged. The user may also answer 'why' and be shown **Trace** as a 'why' explanation. In addition the user should be allowed to ask: What is currently the probability of my hypothesis? The user may then, if tired of supplying information (or pressed by time), conclude the consultation session with a system's answer that is based on partial information only.

```
% Procedure
%
% explore( Goal, Trace, Answer)
%
% finds a likelihood measure that Goal is true. Answer
% contains this likelihood. Trace is the chain of ancestor
% goals and rules, and can be used for 'why' explanations.

explore( Goal, Trace, ( Goal : Prob) was 'derived by' RulesAnswers) :-
  bagof( Rule : if Condition then Goal with Strength,
         Rule : if Condition then Goal with Strength,
         Rules),                                    % All rules about Goal
prior( Goal, Prob0),                                % Prior probability of Goal
modify( Prob0, Rules, Trace, Prob, RulesAnswers).   % Modify prior probability

explore( Goal1 and Goal2, Trace,
        ( Goal1 and Goal2 : P) was 'derived from' ( Answer1 and Answer2) ) :-
  !,
  explore( Goal1, Trace, Answer1),
  explore( Goal2, Trace, Answer2),
  probability( Answer1, P1),
  probability( Answer2, P2),
  min( P1, P2, P).
```

```
explore( Goal1 or Goal2, Trace,
        ( Goal1 or Goal2 : P) was 'derived from' ( Answer1 and Answer2) ) :-
  !,
  explore( Goal1, Trace, Answer1),
  explore( Goal2, Trace, Answer2),
  probability( Answer1, P1),
  probability( Answer2, P2),
  max( P1, P2, P).

explore( not Goal, Trace, ( not Goal : Prob) was 'derived from' Answer) ) :-
  !,
  explore( Goal, Trace, Answer),
  probability( Answer, P),
  invert( P, Prob).

explore( Goal, Trace, ( Goal : Prob) was told) :-
  useranswer( Goal, Trace, Prob).                    % User-supplied answer


% Relation
%
% modify( Prob0, Rules, Trace, Prob, RulesAnswers)
%
% There is a goal Goal whose prior probability is Prob0; Rules
% bear on Goal; the cumulative effect of these rules (through
% their condition parts) modifies Prob0 into Goal's posterior
% probability Prob; Trace is the list of Goal's ancestor goals
% and rules; RulesAnswers are the results of analysis of the
% condition parts of Rules.


modify( Prob0, [], Trace, Prob0, [] ).              % No rule – no effect

modify( Prob0,
        [Rule : if Cond then Goal with Strength | Rules],
        Trace, Prob, [Rule from Answer | RulesAnswers] )
  :-
  explore( Cond, [Goal by Rule | Trace], Answer),   % Condition of first rule
  prior( Cond, P0),
  probability( Answer, P),
  implies( P0, P, Strength, Prob0, Prob1),          % A 'soft implication' rule
  modify( Prob1, Rules, Trace, Prob, RulesAnswers).
```

Figure 14.16 Finding the likelihood of a hypothesis by certainty propagation in an inference network of the Prospector–AL/X type.

14.7 Concluding remarks

Our expert system shell can be elaborated in a number of ways. Several critical comments and suggestions for elaboration can be made at this point.

Our programs are a straightforward implementation of basic ideas, and do not pay much attention to the issue of efficiency. A more efficient implementation would require more sophisticated data structures, indexing or hierarchy of rules, etc.

Our **explore** procedure is susceptible to cycling if the rules in the knowledge base 'cyclicly' mention the same goal. This can be easily rectified by adding a cycle check in **explore**: test whether the current goal is an instance of another goal that is already on **Trace**.

Our 'how' explanation outputs a whole proof tree. In the case of a large proof tree it would be better to output just the top part of the tree and then let the user 'walk' through the rest of the tree as he or she wishes. The user would then inspect the proof tree selectively by using commands such as 'Move down branch 1', 'Move down branch 2', ..., 'Move up', 'Enough'.

In the 'how' and 'why' explanations, our shell just mentions rules by their names, and does not show the rules explicitly. The user should be offered the option to request rules to be displayed explicitly during a consultation session.

Querying the user so that the dialogue looks natural proved to be complicated. Our solution works to some extent, but further problems may appear in several ways, for example:

Is it true: susan flies?
no.

Is it true: susan isa good flyer?

Of course not, if Susan cannot fly at all! Another example is:

Any (more) solution to: Somebody flies?
yes.
Somebody = **bird.**
Is it true: albatross flies?

To cope with such defects, additional relations between concepts dealt with by the expert system would have to be added. Typically, these new relations would specify hierarchical relations between objects and properties

Another refinement of the user-querying procedure would involve the planning of an optimal querying strategy. The optimization objective would be to minimize the number of questions asked of the user before a conclusion is reached. There would be, of course, alternative strategies and which of them would eventually be the shortest would depend on user's answers. A decision of what alternative strategy to pursue can be based on some *a priori* probabilities to assess probabilistically the 'cost' of each alternative. This assessment might have to be recomputed after each user's answer.

There is another measure that can be optimized: the length of the derivation of a conclusion. This would tend to produce simple 'how' explanations. We can reduce the complexity of explanations also by selectively treating individual rules. Thus some rules would not be put into **Trace** and **Answer** in the **explore** procedure. In this case the knowledge base would have to specify which rules are 'traceable', and should therefore appear in explanations, and which should not.

An intelligent expert system should be probabilistically guided so that it concentrates on the currently most likely hypothesis among the competing ones. It should query the user about the information that discriminates best among the top hypotheses.

Our example expert systems were of classification, or 'analysis', type as opposed to the 'synthesis' type where the task is to *construct* something. The result can in the latter case be a plan of actions to accomplish some task – for example, a plan for a robot, a computer configuration that satisfies a given specification, or a forced combination in chess. Our fault diagnosis example can be naturally extended to involve actions; for example, if nothing can be inferred because devices are switched off the system may suggest 'Switch on light 3'. This would entail the problem of optimal plans: minimize the number of actions necessary to reach a conclusion.

Projects

Complete our shell that deals with uncertainties (add a corresponding **useranswer** and other procedures).

Consider critical comments and possible extensions to our expert system shell, as discussed, and design and implement corresponding improvements.

Summary

- Typical functions that are required of an expert system are:

  solving problems in a given domain,
  explaining the problem-solving process,
  dealing with uncertainty and incomplete information.

- It is convenient to view an expert system as consisting of two modules: a shell and a knowledge base. A shell, in turn, consists of an inference mechanism and a user interface.
- Building an expert system shell involves decisions regarding the knowledge-representation formalism, the inference mechanism, the user-interaction facility and the treatment of uncertainty.
- If-then rules, or production rules, are the most common form of representing knowledge in expert systems.

- The shell, developed and programmed in this chapter,

  interprets if-then rules,
  provides 'how' and 'why' explanations, and
  queries the user about the information needed.

- The inference engine of our shell was extended to handle uncertainty.
- Concepts discussed in this chapter are:

  expert systems
  knowledge base, expert system shell, inference engine
  if-then rules, production rules
  'how' explanation, 'why' explanation
  categorical knowledge, uncertain knowledge
  inference network, certainty propagation

References

A collection of papers that deal with various aspects of expert systems and knowledge engineering is Michie (1979). Two early and influential expert systems, MYCIN and Prospector, are described by Shortliffe (1976) and Duda *et al.* (1979). Buchanan and Shortliffe (1984) is a good collection of papers related to the MYCIN experiments. Weiss and Kulikowski (1984) describe practical experience in designing expert systems. The question of handling uncertainty in expert systems is not quite settled yet; Quinlan (1983) compares various approaches. The design of our expert system shell is to some degree similar to that described by Hammond (1984). Some of the examples used in the text are adapted from Winston (1984), Shortliffe (1976), Duda *et al.* (1979), Bratko (1982), and Reiter (1980).

Bratko, I. (1982) Knowledge-based problem-solving in AL3. In: *Machine Intelligence 10* (J. E. Hayes, D. Michie, Y. H. Pao, eds.). Ellis Horwood.

Buchanan, B. G. and Shortliffe, E. H. (1984, eds.) *Rule-based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project.* Addison-Wesley.

Duda, R., Gaschnig, J. and Hart, P. (1979) Model design in the Prospector consultant system for mineral exploration. In: *Expert Systems in the Microelectronic Age* (D. Michie, ed.). Edinburgh University Press.

Hammond, P. (1984) Micro-PROLOG for Expert Systems. In: *Micro-PROLOG: Programming in Logic* (K. L. Clark, F. G. McCabe, eds.). Prentice-Hall.

Michie, D. (1979, ed.) *Expert Systems in the Microelectronic Age.* Edinburgh University Press.

Quinlan, J. R. (1983) Inferno: a cautious approach to uncertain reasoning. *The Computer Journal* **26**: 255–270.

Reiter, J. (1980) AL/X: An Expert System Using Plausible Inference. Oxford: Intelligent Terminals Ltd.

Shortliffe, E. (1976) *Computer-based Medical Consultations: MYCIN*. Elsevier.

Weiss, S. M. and Kulikowski, C. A. (1984) *A Practical Guide to Designing Expert Systems*. Chapman and Hall.

Winston, P. H. (1984) *Artificial Intelligence* (second edition). Addison-Wesley.

15 Game Playing

In this chapter we will consider techniques for playing two-person, perfect-information games, such as chess. For interesting games, trees of possible continuations are far too complex to be searched exhaustively, so other approaches are necessary. One method is based on the minimax principle, efficiently implemented as the alpha-beta algorithm. In addition to this standard technique, we will develop in this chapter a program based on the Advice Language approach for introducing pattern knowledge into a chess-playing program. This rather detailed example further illustrates how well Prolog is suited for the implementation of knowledge-based systems.

15.1 Two-person, perfect-information games

The kind of games that we are going to discuss in this chapter are called two-person, perfect-information games. Examples of games of this kind are chess, checkers and go. In such games there are two players that make moves alternatively, and both players have the complete information of the current situation in the game. Thus this definition excludes most card games. The game is over when a position is reached that qualifies as 'terminal' by the rules of the game – for example, mate in chess. The rules also determine what is the outcome of the game that has ended in this terminal position.

Such a game can be represented by a *game tree*. The nodes in such a tree correspond to situations, and the arcs correspond to moves. The initial situation of the game is the root node; leaves of the tree correspond to terminal positions.

In most games of this type the outcome of the game can be *win*, *loss* or *draw*. We will now consider games with just two outcomes: *win* and *loss*. Games where a draw is a possible outcome can be reduced to two outcomes: *win*, *not-win*. The two players will be called 'us' and 'them'. 'Us' can win in a non-terminal 'us-to-move' position if there is *a* legal move that leads to a won position. On the other hand, a non-terminal 'them-to-move' position is won for 'us' if *all* the legal moves from this position lead to won positions. These rules correspond to AND/OR tree representation of problems discussed in Chapter

13. The concepts from AND/OR trees and games correspond as follows:

game positions	problems
terminal won position	goal node, trivially solved problem
terminal lost position	unsolvable problem
won position	solved problem
us-to-move position	OR node
them-to-move position	AND node

Clearly, many concepts from searching AND/OR trees can be adapted for searching game trees.

A simple program that finds whether an us-to-move position is won can be defined as follows:

```
won( Pos) :-
  terminalwon( Pos).                    % A terminal won position

won( Pos) :-
  not terminallost( Pos),
  move( Pos, Pos1),                     % A legal move to Pos1
  not ( move( Pos1, Pos2),              % No opponent's move leads to a
        not won( Pos2) ).               % not-won position
```

The rules of the game are built into the predicates **move(Pos, Pos1)** to generate legal moves, and **terminalwon(Pos)** and **terminallost(Pos)** to recognize terminal positions that are won or lost by the rules of the game. The last rule above says, through the double use of **not**: there is no them-move that leads to a not-won position. In other words: *all* them-moves lead to a won position.

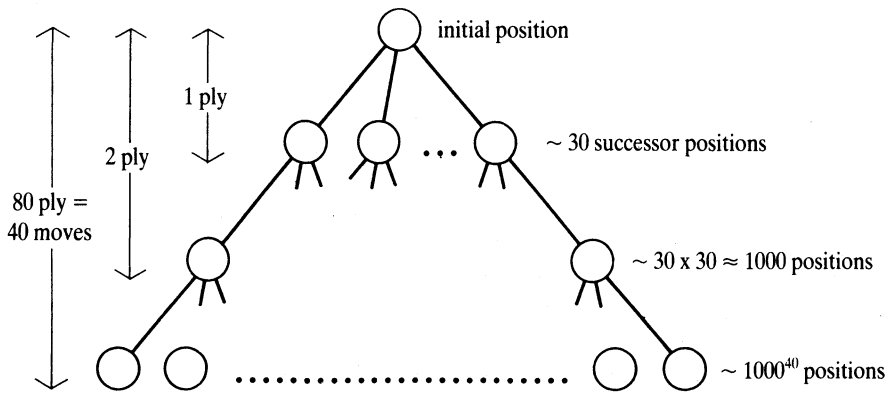


Figure 15.1 The complexity of game trees in chess. The estimates here are based on an approximation that there are about 30 legal moves from each chess position, and that terminal positions occur at a depth of 40 moves. One move is 2 plies (1 half-move by each side).

As with analogous programs for searching AND/OR graphs, the above program uses the depth-first strategy. In addition, this program does not prevent cycling between positions. This may cause problems as the rules of some games allow repetition of positions. However, this repetition is often only superficial. By rules of chess, for example, after a three-fold repetition the game can be claimed a draw.

The foregoing program shows the basic principle. However, much more powerful techniques are necessary for practically dealing with complicated games like chess or go. The combinatorial complexity of these games makes our naive search algorithm, which only stops at terminal positions of the game, completely infeasible. Figure 15.1 illustrates this point with respect to chess. The search space of astronomical proportions includes some 10^{120} positions. It can be argued that equal positions in the tree of Figure 15.1 occur at different places. Still, it has been shown that the number of different positions is far beyond anything manageable by forseeable computers.

Project

Write a program to play some simple game (like *nim*) using the straightforward AND/OR search approach.

15.2 The minimax principle

As searching game trees exhaustively is not feasible for interesting games, other methods that rely on searching only part of the game tree have been developed. Among these, a standard technique used in computer game playing (chess) is based on the *minimax* principle. A game tree is only searched up to a certain depth, typically a few moves, and then the tip nodes of the search tree are evaluated by some evaluation function. The idea is to assess these terminal search positions without searching beyond them, thus saving time. These terminal position estimates then propagate up the search tree according to the minimax principle. This yields position values for all the positions in the search tree. The move that leads from the initial, root position to its most promising successor (according to these values) is then actually played in the game.

Notice that we distinguish between a 'game tree' and a 'search tree'. A search tree is normally a part of the game tree (upper part) – that is, the part that is explicitly generated by the search process. Thus, terminal search positions do not have to be terminal positions of the game.

Much depends on the evaluation function which, in most games of interest, has to be a heuristic estimator that estimates the winning chances from the point of view of one of the players. The higher the value the higher the player's chances are to win, and the lower the value the higher the opponent's chances are to win. As one of the players will tend to achieve a high position value, and the other a low value, the two players will be called MAX and MIN

respectively. Whenever MAX is to move, he or she will choose a move that maximizes the value; on the contrary, MIN will choose a move that minimizes the value. Given the values of the bottom-level positions in a search tree, this principle (called *minimax*) will determine the values of all the other positions in the search tree. Figure 15.2 illustrates. In the figure, levels of positions with MAX to move alternate with those with MIN to move. The bottom-level position values are determined by the evaluation function. The values of the internal nodes can be computed in a bottom-up fashion, level by level, until the root node is reached. The resulting root value in Figure 15.2 is 4, and accordingly the best move for MAX in position *a* is *a-b*. The best MIN's reply is *b-d*, etc. This sequence of play is also called the *main variation*. The main variation defines the 'minimax-optimal' play for both sides. Notice that the value of the positions along the main variation does not vary. Accordingly, correct moves are those that preserve the value of the game.

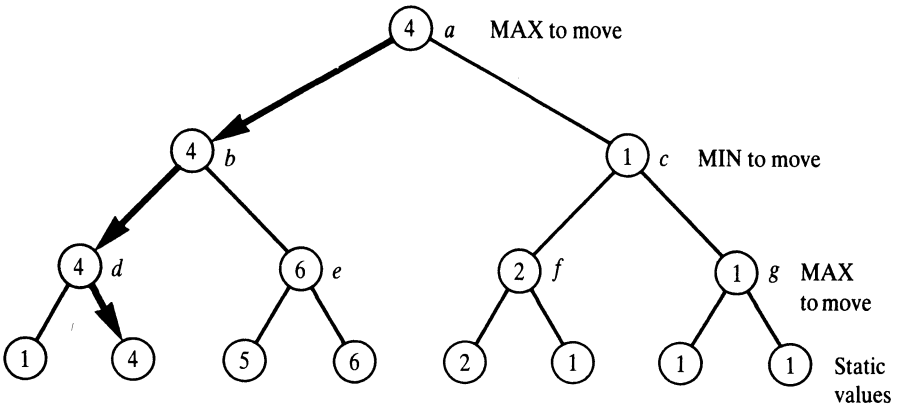


Figure 15.2 Static values (bottom level) and minimax backed-up values in a search tree. The indicated moves constitute the *main variation* – that is, the minimax optimal play for both sides.

We distinguish between the bottom-level values and the backed-up values. The former values are called 'static' since they are obtained by a 'static' evaluation function, as opposed to backed-up values that are obtained 'dynamically' by propagation of static values up the tree.

The value propagation rules can be formalized as follows. Let us denote the static value of a position *P* by

$v(P)$

and the backed-up value by

$V(P)$

Let $P_1, \ldots, P_n$ be legal successor positions of P. Then the relation between static values and backed-up values can be defined as:

$$V(P) = v(P) \quad \text{if } P \text{ is a terminal position in a search tree } (n = 0)$$
$$V(P) = \max_i V(P_i) \quad \text{if } P \text{ is a MAX-to-move position}$$
$$V(P) = \min_i V(P_i) \quad \text{if } P \text{ is a MIN-to-move position}$$

A Prolog program that computes the minimax backed-up value for a given position is shown in Figure 15.3. The main relation in this program is

minimax(Pos, BestSucc, Val)

where **Val** is the minimax value of a position **Pos**, and **BestSucc** is the best successor position of **Pos** (the move to be played to achieve **Val**). The relation

moves(Pos, PosList)

corresponds to the legal-move rules of the game: **PosList** is the list of legal

```
% Minimax procedure: minimax( Pos, BestSucc, Val)
% Pos is a position, Val is its minimax value; best move
% from Pos leads to position BestSucc

minimax( Pos, BestSucc, Val) :-
  moves( Pos, PosList), !,                % Legal moves in Pos produce PosList
  best( PosList, BestSucc, Val);
  staticval( Pos, Val).                   % Pos has no successors

best( [Pos], Pos, Val) :-
  minimax( Pos, _, Val), !.

best( [Pos1 | PosList], BestPos, BestVal) :-
  minimax( Pos1, _, Val1),
  best( PosList, Pos2, Val2),
  betterof( Pos1, Val1, Pos2, Val2, BestPos, BestVal).

betterof( Pos0, Val0, Pos1, Val1, Pos0, Val0) :-
  min_to_move( Pos0), Val0 > Val1, !;
  max_to_move( Pos0), Val0 < Val1, !.

betterof( Pos0, Val0, Pos1, Val1, Pos1, Val1).
```

Figure 15.3 A straightforward implementation of the minimax principle.

successor positions of **Pos**. The predicate **moves** is assumed to fail if **Pos** is a terminal search position (a leaf of the search tree). The relation

best(PosList, BestPos, BestVal)

selects the 'best' position **BestPos** from a list of candidate positions **PosList**. **BestVal** is the value of **BestPos**, and hence also of **Pos**. 'Best' is here either maximum or minimum, depending on the side to move.

15.3 The alpha-beta algorithm: an efficient implementation of minimax

The program in Figure 15.3 systematically visits *all* the positions in the search tree, up to its terminal positions in a depth-first fashion, and statically evaluates *all* the terminal positions of this tree. Usually not all this work is necessary in order to correctly compute the minimax value of the root position. Accordingly, the search algorithm can be economized. The improvement can be based on the following idea: Suppose that there are two alternative moves; once one of them has been shown to be clearly inferior to the other, it is not necessary to know *exactly* how much inferior it is for making the correct decision. For example, we can use this principle to reduce the search in the tree of Figure 15.2. The search process here proceeds as follows:

(1) Start with position a.
(2) Move down to b.
(3) Move down to d.
(4) Take the maximum of d's successors yielding $V(d) = 4$.
(5) Backtrack to b and move down to e.
(6) Consider the first successor of e whose value is 5. At this point MAX (who is to move in e) is guaranteed at least the value of 5 in position e regardless of other (possibly better) alternatives from e. This is sufficient for MIN to realize that, at node b, the alternative e is inferior to d, even without knowing the exact value of e.

On these grounds we can neglect the second successor of e and simply assign to e an *approximate* value 5. This approximation will, however, have no effect on the value of b and, hence, of a.

The celebrated *alpha-beta algorithm* for efficient minimaxing is based on this idea. Figure 15.4 illustrates the action of the alpha-beta algorithm on our example tree of Figure 15.2. As Figure 15.4 shows, some of the backed-up values are approximate. However, these approximations are sufficient to

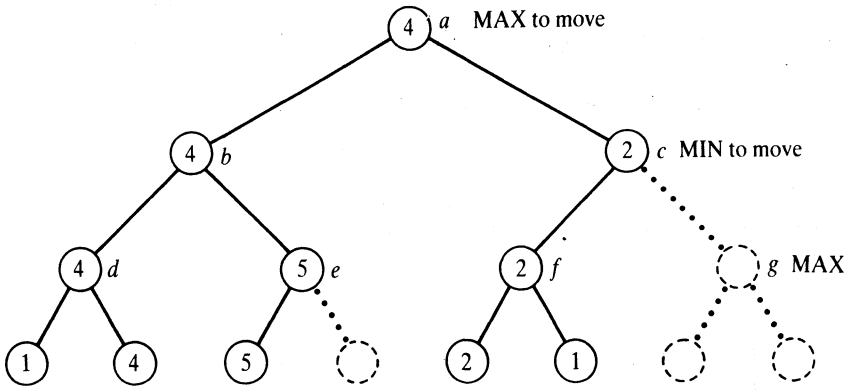


Figure 15.4 The tree of Figure 15.2 searched by the alpha-beta algorithm. The alpha-beta search prunes the nodes shown by dotted lines, thus economizing the search. As a result, some of the backed-up values are inexact (nodes *c, e, f*; compare with Figure 15.2). However, these approximations suffice for determining the root value and the main variation correctly.

determine the root value precisely. In the example of Figure 15.4, the alpha-beta principle reduces the search complexity from eight static evaluations (as originally in Figure 15.2) to five static evaluations.

As said before, the key idea of the alpha-beta pruning is to find a 'good enough' move, not necessarily the best, that is sufficiently good to make the correct decision. This idea can be formalized by introducing two bounds, usually denoted *Alpha* and *Beta*, on the backed-up value of a position. The meaning of these bounds is: *Alpha* is the minimal value that MAX is already guaranteed to achieve, and *Beta* is the maximal value that MAX can hope to achieve. From MIN's point of view, *Beta* is the worst value for MIN that MIN is guaranteed to achieve. Thus, the actual value (that is to be found) lies between *Alpha* and *Beta*. If a position has been shown that its value lies outside the *Alpha-Beta* interval then this is sufficient to know that this position is not in the main variation, without knowing the exact value of this position. We only have to know the exact value of this position if this value is between *Alpha* and *Beta*. Formally, we can define a 'good enough' backed-up value *V(P,Alpha,Beta)* of a position *P*, with respect to *Alpha* and *Beta*, as any value that satisfies the following requirements:

$$
\begin{aligned}
&V(P,Alpha,Beta) \leqq Alpha \quad &&if \quad &&V(P) \leqq Alpha \\
&V(P,Alpha,Beta) = V(P) \quad &&if \quad &&Alpha < V(P) < Beta \\
&V(P,Alpha,Beta) \geqq Beta \quad &&if \quad &&V(P) \geqq Beta
\end{aligned}
$$

Obviously we can always compute the exact value of a root position *P* by setting the bounds as follows:

$$V(P, -infinity, +infinity) = V(P)$$

Figure 15.5 shows a Prolog implementation of the alpha-beta algorithm. The main relation is

alphabeta(Pos, Alpha, Beta, GoodPos, Val)

where **GoodPos** is a 'good enough' successor of **Pos**, so that its value **Val** satisfies the requirements stated above:

Val = *V(Pos,Alpha,Beta)*

```
% The alpha-beta algorithm

alphabeta( Pos, Alpha, Beta, GoodPos, Val) :-
  moves( Pos, PosList), !,
  boundedbest( Poslist, Alpha, Beta, GoodPos, Val);
  staticval( Pos, Val).

boundedbest( [Pos | PosList], Alpha, Beta, GoodPos, GoodVal) :-
  alphabeta( Pos, Alpha, Beta, _, Val),
  goodenough( PosList, Alpha, Beta, Pos, Val, GoodPos, GoodVal).

goodenough( [], _, _, Pos, Val, Pos, Val) :- !.      % No other candidate

goodenough( _, Alpha, Beta, Pos, Val, Pos, Val) :-
  min_to_move( Pos), Val > Beta, !;          % Maximizer attained upper bound
  max_to_move( Pos), Val < Alpha, !.         % Minimizer attained lower bound

goodenough( PosList, Alpha, Beta, Pos, Val, GoodPos, GoodVal) :-
  newbounds( Alpha, Beta, Pos, Val, NewAlpha, NewBeta), % Refine bounds
  boundedbest( PosList, NewAlpha, NewBeta, Pos1, Val1),
  betterof( Pos, Val, Pos1, Val1, GoodPos, GoodVal).

newbounds( Alpha, Beta, Pos, Val, Val, Beta) :-
  min_to_move( Pos), Val > Alpha, !.         % Maximizer increased lower bound

newbounds( Alpha, Beta, Pos, Val, Alpha, Val) :-
  max_to_move( Pos), Val < Beta, !.          % Minimizer decreased upper bound

newbounds( Alpha, Beta, _, _, Alpha, Beta).

betterof( Pos, Val, Pos1, Val1, Pos, Val) :-
  min_to_move( Pos), Val > Val1, !;
  max_to_move( Pos), Val < Val1, !.

betterof( _, _, Pos1, Val1, Pos1, Val1).
```

Figure 15.5 An implementation of the alpha-beta algorithm.

The procedure

boundedbest(PosList, Alpha, Beta, GoodPos, Val)

finds a good enough position **GoodPos** in the list **PosList** so that the backed-up value **Val** of **GoodPos** is a good enough approximation with respect to **Alpha** and **Beta**.

The alpha-beta interval may get narrower (but never wider!) at deeper recursive calls of the alpha-beta procedure. The relation

newbounds(Alpha, Beta, Pos, Val, NewAlpha, NewBeta)

defines the new interval (**NewAlpha, NewBeta**). This is always narrower than or equal to the old interval (**Alpha, Beta**). So at deeper levels in the search tree, the *Alpha-Beta* bounds tend to shrink, and positions at deeper levels are evaluated with tighter bounds. Narrower intervals allow for grosser approximations, and thus more tree pruning. An interesting question is now: How much effort the alpha-beta algorithm saves compared with the exhaustive minimax search program of Figure 15.3?

The efficiency of the alpha-beta search depends on the order in which positions are searched. It is advantageous to consider strong moves for each side first. It is easy to demonstrate by examples that if the order is unfortunate then the alpha-beta procedure will have to visit *all* the positions visited by the exhaustive minimax search. That means that in the worst case alpha-beta will have no advantage over the exhaustive minimax search. If the order is favourable, however, savings can be significant. Let N be the number of terminal search positions statically evaluated by the exhaustive minimax algorithm. It has been proved that in the best case, when the strongest move is always considered first, the alpha-beta algorithm will only have to statically evaluate $\sqrt{N}$ positions.

On a similar note, this same result is relevant in a practical aspect in tournament play. In a tournament, a chess-playing program is usually given a certain amount of time for computing the next move in the game, and the depth to which the program can search will depend on this amount of time. The alpha-beta algorithm will be able, in the best case, to search *twice as deep* as the exhaustive minimax search. Experience shows that the same evaluation function applied at a greater depth in the tree will usually produce stronger play.

The economization effect of the alpha-beta algorithm can also be expressed in terms of the effective branching factor (number of branches stemming from each internal node) of the search tree. Assume that the game tree has a uniform branching factor b. Due to the pruning effect, alpha-beta will only search some of the branches, thus effectively reducing the branching factor. The reduction is, in the best case, from b to $\sqrt{b}$. In chess-playing programs the effective branching factor due to the alpha-beta pruning becomes about 6 compared to the total of about 30 legal moves. A less optimistic view on this result is that in chess, even with alpha-beta, deepening the search by 1 ply

(one halph-move) increases the number of terminal search positions by a factor of about 6.

Project

Consider a two-person game (for example, some non-trivial version of tic-tac-toe). Write game-definition relations (legal moves and terminal game positions) and propose a static evaluation function to be used for playing the game with the alpha-beta procedure.

15.4 Minimax-based programs: refinements and limitations

The minimax principle, together with the alpha-beta algorithm, is the basis of many successful game-playing programs, most notably chess programs. The general scheme of such a program is: perform the alpha-beta search on the current position in the game, up to some fixed depth limit (dictated by the time constraints imposed by tournament rules), using a game-specific evaluation function for evaluating the terminal positions of the search. Then execute the best move (according to alpha-beta) on the play board, accept the opponent's reply, and start the same cycle again.

The two basic ingredients, then, are the alpha-beta algorithm and a heuristic evaluation function. To build a good program for a complicated game like chess many refinements to this basic scheme are needed. We will briefly review some standard techniques.

Much depends on the evaluation function. If we had a perfect evaluation function we would only have to consider the immediate successors of the current position, thus practically eliminating search. But for games like chess, any evaluation function of practically acceptable computational complexity will necessarily be just a heuristic estimate. This estimate is based on 'static' features of the position (for example, the number of pieces on the board) and will therefore be more reliable in some positions than in others. Consider for example such a material-based evaluation function for chess and imagine a position in which White is a knight up. This function will, of course, assess the position in White's favour. This is fine if the position is quiescent, Black having no violent threat at his disposal. On the other hand, if Black can capture the White's queen on the next move, such an evaluation can result in a disastrous blunder, as it will not be able to perceive the position *dynamically*. Clearly, we can better trust the static evaluation in quiescent positions than in turbulent positions in which each side has direct threats of capturing the opponent's pieces. Obviously, we should use the static evaluation only in quiescent positions. Therefore a standard trick is to extend the search in turbulent positions beyond the depth limit until a quiescent position is reached. In particular, this extension includes sequences of piece captures in chess.

Another refinement is *heuristic pruning*. This aims at achieving a greater depth limit by disregarding some less promising continuations. This technique will prune branches in addition to those that are pruned by the alpha-beta technique itself. Therefore this entails the risk of overlooking some good continuation and incorrectly computing the minimax value.

Yet another technique is *progressive deepening*. The program repeatedly executes the alpha-beta search, first to some shallow depth, and then increases the depth limit on each iteration. The process stops when the time limit has been reached. The best move according to the deepest search is then played. This technique has the following advantages:

- enables the time control; when the time limit is reached there will always be some best move found so far;
- the minimax values of the previous iteration can be used for preliminary ordering of position on the next iteration, thus helping the alpha-beta algorithm to search strong moves first.

Progressive deepening entails some overhead (researching upper parts of the game tree), but this is relatively small compared with the total effort.

A known problem with programs that belong to this general scheme is the 'horizon effect'. Imagine a chess position in which the program's side inevitably loses a knight. But the loss of the knight can be delayed at the cost of a lesser sacrifice, a pawn say. This intermediate sacrifice may push the actual loss of the knight beyond the search limit (beyond the program's 'horizon'). Not seeing the eventual loss of the knight, the program will then prefer this variation to the quick death of the knight. So the program will eventually lose *both* the pawn (unnecessarily) and the knight. The extension of search up to a quiescent position can alleviate the horizon effect.

There is, however, a more fundamental limitation of the minimax-based programs which lies in the limited form of the domain-specific knowledge they use. This becomes very conspicuous when we compare the best chess programs with human chess masters. Strong programs often search millions (and more) of positions before deciding on the move to play. It is known from psychological studies that human masters typically search just a few tens of positions, at most a few hundred. Despite this apparent inferiority, chess masters usually beat programs without too much effort. The masters' advantage lies in their knowledge, which far exceeds that contained in the programs. Games between machines and strong human players show that the enormous advantage in the calculating power cannot completely compensate the lack of knowledge.

Knowledge in minimax-based programs takes three main forms:

- evaluation function,
- tree-pruning heuristics,
- quiescence heuristics.

The evaluation function reduces many aspects of a game situation into a single

number, and this reduction can have a detrimental effect. A good player's understanding of a game position, on the contrary, spans over many dimensions. Let us consider an example from chess: an evaluation function will evaluate a position as equal simply by stating that its value is 0. A master's assessment of the same position can be much more informative and indicative of a further course of the game. For example, Black is a pawn up, but White has a good attacking initiative that compensates the material, so chances are equal.

In chess, minimax-based programs often play well in sharp tactical struggles when precise calculation of forced variations is decisive. Their weakness shows in quiet positions where their play falls short of long-range plans that prevail in such slow, strategic games. Lack of a plan makes an impression that the program keeps wandering during the game from one idea to another. This is particularly evident in chess endgames.

In the rest of this chapter we will consider another approach to game playing, based on introducing pattern knowledge into a program by means of 'advice'.

15.5 Pattern knowledge and the mechanism of 'advice'

15.5.1 Goals and move-constraints

The method of representing game-specific knowledge that we consider in this section belongs to the family of Advice Languages. In Advice Languages the user specifies, in a declarative way, what ideas should be tried in certain types of situations. Ideas are formulated in terms of goals and means of achieving the goals. An Advice Language interpreter then finds out, through search, which idea actually works in a given situation.

The fundamental concept in Advice Languages is a 'piece-of-advice'. A piece-of-advice suggests what to do (or to *try* to do) next in a certain type of position. Generally speaking, advice is expressed in terms of *goals* to be achieved, and *means* of achieving these goals. The two sides are called 'us' and 'them'; advice always refers to the 'us' point of view. Each piece-of advice has four ingredients:

- *better-goal:* a goal to be achieved;
- *holding-goal:* a goal to be maintained during play toward the better-goal;
- *us-move-constraints:* a predicate on moves that selects a subset of all legal us-moves (moves that should be considered of interest with respect to the goals specified);
- *them-move-constraints:* a predicate to select moves to be considered by 'them' (moves that may undermine the goals specified).

As a simple example from the chess endgame king and pawn vs. king, consider

the straightforward idea of queening the pawn by simply pushing the pawn forward. This can be expressed in the form of advice as:

- *better-goal:* pawn queened;
- *holding-goal:* pawn is not lost;
- *us-move-constraints:* pawn move;
- *them-move-constraints:* approach the pawn with the king.

15.5.2 Satisfiability of advice

We say that a given piece-of-advice is *satisfiable* in a given position if 'us' can force the achievement of the better-goal specified in the advice under the conditions that:

(1) the holding-goal is never violated,
(2) all the moves played by 'us' satisfy us-move-constraints,
(3) 'them' is only allowed to make moves that satisfy them-move-constraints.

The concept of a *forcing-tree* is associated with the satisfiability of a piece-of-advice. A forcing-tree is a detailed strategy that guarantees the achievement of the better-goal under the constraints specified by the piece-of-advice. A forcing-tree thus specifies exactly what moves 'us' has to play on any 'them' reply. More precisely, a forcing-tree T for a given position P and a piece-of-advice A is a subtree of the game tree such that:

- the root node of T is P;
- all the positions in P satisfy the holding-goal;
- all the terminal nodes in T satisfy the better-goal, and no internal node in T satisfies the better-goal;
- there is exactly one us-move from each internal us-to-move position in T; and that move must satisfy the us-move-constraints
- T contains all them-moves (that satisfy the them-move-constraints) from each non-terminal them-to-move position in T.

Each piece-of-advice can be viewed as a definition of a small special game with the following rules. Each opponent is allowed to make moves that satisfy his or her move-constraints; a position that does not satisfy the holding-goal is won for 'them'; a position that satisfies the holding-goal and the better-goal is won for 'us'. A non-terminal position is won for 'us' if the piece-of-advice is satisfiable in this position. Then 'us' will win by executing a corresponding forcing-tree in the play.

15.5.3 Integrating pieces-of-advice into rules and advice-tables

In Advice Languages, individual pieces-of-advice are integrated in the complete knowledge representation schema through the following hierarchy. A piece-of-advice is part of an if-then rule. A collection of if-then rules is an *advice-table*. A set of advice-tables is structured into a hierarchical network. Each advice-table has the role of a specialized expert to deal with some specific subproblem of the whole domain. An example of such a specialized expert is an advice-table that knows how to mate in the king and rook vs. king ending in chess. This table is summoned when such an ending occurs in a game.

For simplicity, we will consider a simplified version of an Advice Language in which we will only allow for one advice-table. We shall call this version Advice Language 0, or AL0 for short. Here is the structure of AL0 already syntactically tailored toward an easy implementation in Prolog.

A program in AL0 is called an *advice-table*. An advice-table is an *ordered* collection of if-then rules. Each rule has the form:

RuleName : if Condition then AdviceList

Condition is a logical expression that consists of predicate names connected by logical connectives **and**, **or**, **not**. **AdviceList** is a list of names of pieces-of-advice. An example of a rule called 'edge_rule', from the king and rook vs. king ending, can be:

```
edge_rule :
  if their_king_on_edge and our_king_close
  then [ mate_in_2, squeeze, approach, keeproom, divide].
```

This rule says: if in the current position their king is on the edge and our king is close to their king (or more precisely, kings are less than four squares apart), then try to satisfy, in the order of preference as stated, the pieces-of-advice: 'mate_in_2', 'squeeze', 'approach', 'keeproom', 'divide'. This advice-list specifies pieces-of-advice in the decreasing order of ambition: first try to mate in two moves, if that is not possible then try to 'squeeze' the opponent's king toward a corner, etc. Notice that with an appropriate definition of operators, the rule above is a syntactically correct Prolog clause.

Each piece-of-advice will be specified by a Prolog clause of another form:

```
advice( AdviceName,
        BetterGoal :
        HoldingGoal :
        Us_Move_Constraints :
        Them_Move_Constraints).
```

The goals are expressions that consist of predicate names and logical connectives **and**, **or**, **not**. Move-constraints are, again, expressions that consist of predicate names and the connectives **and** and **then**: **and** has the usual logical

meaning, **then** prescribes the ordering. For example, a move-constraint of the form

MC1 then MC2

says: first consider those moves that satisfy **MC1**, and then those that satisfy **MC2**.

For example, a piece-of-advice to mate in 2 moves in the king and rook vs. king ending, written in this syntax, is:

```
advice( mate_in_2,
        mate :
        not rooklost :
        (depth = 0) and legal then (depth = 2) and checkmove :
        (depth = 1) and legal).
```

Here the better-goal is **mate**, the holding-goal is **not rooklost** (rook is not lost). The us-move-constraints say: at depth 0 (the current board position) try any legal move, then at depth 2 (our second move) try checking moves only. The depth is measured in plies. Them-move-constraints are: any legal move at depth 1.

In playing, an advice-table is then used by repeating, until the end of the game, the following main cycle: build a forcing-tree, then play according to this tree until the play exits the tree; build another forcing-tree, etc. A forcing-tree is generated each time as follows: take the current board position **Pos** and scan the rules in the advice-table one by one; for each rule, match **Pos** with the precondition of the rule, and stop when a rule is found such that **Pos** satisfies its precondition. Now consider the advice-list of this rule: process pieces-of-advice in this list one by one until a piece-of-advice is found that is satisfiable in **Pos**. This results in a forcing-tree that is the detailed strategy to be executed across the board.

Notice the importance of the ordering of rules and pieces-of-advice. The rule used is the first rule whose precondition matches the current position. There must be for any possible position at least one rule in the advice-table whose precondition will match the position. Thus an advice-list is selected. The first piece-of-advice in this list that is satisfiable is applied.

An advice-table is thus largely a non-procedural program. An AL0 interpreter accepts a position and by executing an advice-table produces a forcing-tree which determines the play in that position.

15.6 A chess endgame program in Advice Language 0

Implementation of an AL0-based game-playing program can be conveniently divided into three modules:

(1) an AL0 interpreter,

(2) an advice-table in AL0,

(3) a library of predicates (including rules of the game) used in the advice-table.

This structure corresponds to the usual structure of knowledge-based systems as follows:

- The AL0 interpreter is an inference engine.
- The advice-table and the predicate library constitute a knowledge base.

15.6.1 A miniature AL0 interpreter

A miniature, game-independent AL0 interpreter is implemented in Prolog in Figure 15.6. This program also performs the user interaction during play. The central function of the program is the use of knowledge in an AL0 advice-table; that is, interpreting an AL0 advice-program for the generation of forcing-trees and their execution in a game. The basic forcing-tree generation algorithm is similar to the best-first search in AND/OR graphs of Chapter 13; a forcing-tree corresponds to an AND/OR solution tree. On the other hand, it also resembles the generation of a solution tree to a user's query in the expert system shell of Chapter 14.

For simplicity, in the program of Figure 15.6 'us' is supposed to be White, and 'them' is Black. The program is started through the procedure

playgame(Pos)

where **Pos** is a chosen initial position of a game to be played. If it is 'them' to move in **Pos** then the program reads a move from the user, otherwise the program consults the advice-table that is attached to the program, generates a forcing-tree and plays its move according to the tree. This continues until the end of the game is reached as specified by the predicate 'end_of_ game' (mate, for example).

A forcing-tree is a tree of moves, represented in the program by the following structure

Move .. [Reply1 .. Ftree1, Reply2 .. Ftree2, ...]

where '..' is an infix operator; **Move** is the first move for 'us'; **Reply1**, **Reply2**, etc. are the possible 'them' replies; and **Ftree1**, **Ftree2**, etc. are forcing-subtrees that correspond to each of the 'them' replies respectively.

15.6.2 An advice-program for the king and rook vs. king ending

A broad strategy for winning with the king and rook against the sole opponent's king is to force the king to the edge, or into a corner if necessary, and

```
% A miniature implementation of Advice Language 0
%
% This program plays a game from a given starting position
% using knowledge represented in Advice Language 0

:- op( 200, xfy, :).
:- op( 220, xfy, ..).
:- op( 185, fx, if).
:- op( 190, xfx, then).
:- op( 180, xfy, or).
:- op( 160, xfy, and).
:- op( 140, fx, not).

playgame( Pos) :-                          % Play a game starting in Pos
  playgame( Pos, nil).                     % Start with empty forcing-tree

playgame( Pos, ForcingTree) :-
  show( Pos),
  ( end_of_game( Pos),                     % End of game?
      write( 'End of game'), nl, !;
    playmove( Pos, ForcingTree, Pos1, ForcingTree1), !,
      playgame( Pos1, ForcingTree1) ).

% Play 'us' move according to forcing-tree

playmove( Pos, Move .. FTree1, Pos1, FTree1) :-
  side( Pos, w),                           % White = 'us'
  legalmove( Pos, Move, Pos1),
  showmove( Move).

% Read 'them' move

playmove( Pos, FTree, Pos1, FTree1) :-
  side( Pos, b),
  write( 'Your move: '),
  read( Move),
  ( legalmove( Pos, Move, Pos1),
      subtree( FTree, Move, FTree1), !;    % Move down forcing-tree
    write( 'Illegal move'), nl,
      playmove( Pos, FTree, Pos1, FTree1) ).

% If current forcing-tree is empty generate a new one

playmove( Pos, nil, Pos1, FTree1) :-
  side( Pos, w),
  resetdepth( Pos, Pos0),                  % Pos0 = Pos with depth 0
  strategy( Pos0, FTree), !,               % Generate new forcing-tree
  playmove( Pos0, FTree, Pos1, FTree1).
```

```
% Select a forcing-subtree corresponding to Move

subtree( FTrees, Move, FTree) :-
  member( Move .. FTree, FTrees), !.

subtree( _, _, nil).

strategy( Pos, ForcingTree) :-                    % Find forcing-tree for Pos
  Rule : if Condition then AdviceList,            % Consult advice-table
  holds( Condition, Pos, _), !,                   % Match Pos against precondition
  member( AdviceName, AdviceList),                % Try pieces-of-advice in turn
  nl, write( 'Trying'), write( AdviceName),
  satisfiable( AdviceName, Pos, ForcingTree), !. % Satisfy AdviceName in Pos

satisfiable( AdviceName, Pos, FTree) :-
  advice( AdviceName, Advice),                    % Retrieve piece-of-advice
  sat( Advice, Pos, Pos, FTree).                  % 'sat' needs two positions for
                                                  % comparison predicates

sat( Advice, Pos, RootPos, FTree) :-
  holdinggoal( Advice, HG),
  holds( HG, Pos, RootPos),                       % Holding-goal satisfied
  sat1( Advice, Pos, RootPos, FTree).

sat1( Advice, Pos, RootPos, nil) :-
  bettergoal( Advice, BG),
  holds( BG, Pos, RootPos), !.                    % Better-goal satisfied

sat1( Advice, Pos, RootPos, Move .. FTrees) :-
  side( Pos, w), !,                               % White = 'us'
  usmoveconstr( Advice, UMC),
  move( UMC, Pos, Move, Pos1),                    % A move satisfying move-constr.
  sat( Advice, Pos1, RootPos, FTrees).

sat1( Advice, Pos, RootPos, FTrees) :-
  side( Pos, b), !,                               % Black = 'them'
  themmoveconstr( Advice, TMC),
  bagof( Move .. Pos1, move( TMC, Pos, Move, Pos1), MPlist),
  satall( Advice, MPlist, RootPos, FTrees).       % Satisfiable in all successors

satall( _, [], _, [] ).

satall( Advice, [Move .. Pos | MPlist], RootPos, [Move .. FT | MFTs] ) :-
  sat( Advice, Pos, RootPos, FT),
  satall( Advice, MPlist, RootPos, MFTs).

% Interpreting holding and better-goals:
% A goal is an AND/OR/NOT combination of predicate names

holds( Goal1 and Goal2, Pos, RootPos) :- !,
  holds( Goal1, Pos, RootPos),
  holds( Goal2, Pos, RootPos).
```

```
holds( Goal1 or Goal2, Pos, RootPos) :- !,
  ( holds( Goal1, Pos, RootPos);
    holds( Goal2, Pos, RootPos) ).

holds( not Goal, Pos, RootPos) :- !,
  not holds( Goal, Pos, RootPos).

holds( Pred, Pos, RootPos) :-
  ( Cond =.. [Pred, Pos];          % Most predicates do not depend on RootPos
    Cond =.. [Pred, Pos, RootPos] ),
  call( Cond).

% Interpreting move-constraints

move( MC1 and MC2, Pos, Move, Pos1) :- !,
  move( MC1, Pos, Move, Pos1),
  move( MC2, Pos, Move, Pos1).

move( MC1 then MC2, Pos, Move, Pos1) :- !,
  ( move( MC1, Pos, Move, Pos1);
    move( MC2, Pos, Move, Pos1) ).

% Selectors for components of piece-of-advice

bettergoal( BG : _, BG).

holdinggoal( BG : HG : _, HG).

usmoveconstr( BG : HG : UMC : _, UMC).

themmoveconstr( BG : HG : UMC : TMC, TMC).

member( X, [X | L] ).

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

Figure 15.6 A miniature implementation of Advice Language 0.

then deliver mate in a few moves. An elaboration of this broad principle is:

> While making sure that stalemate is never created or the rook left undefended under attack, repeat until mate:
>
> (1) Look for a way to mate the opponent's king in two moves.
>
> (2) If the above is not possible, then look for a way to constrain further the area on the chessboard to which the opponent's king is confined by our rook.
>
> (3) If the above is not possible, then look for a way to move our king closer to the opponent's king.

(4) If none of the above pieces-of-advice 1, 2 or 3 works, then look for a way of maintaining the present achievements in the sense of 2 and 3 (that is, make a waiting move).

(5) If none of 1, 2, 3 or 4 is attainable, then look for a way of obtaining a position in which our rook divides the two kings either vertically or horizontally.

These principles are implemented in detail as an AL0 advice-table in Figure 15.7. This table can be run by the AL0 interpreter of Figure 15.6. Figure 15.8 illustrates the meaning of some of the predicates used in the table and the way the table works.

The predicates used in the table are:

Goal predicates

mate	their king mated
stalemate	their king stalemated
rooklost	their king can capture our rook
rookexposed	their king can attack our rook before our king can get to defend the rook
newroomsmaller	area to which their king is restricted by our rook has shrunk
rookdivides	rook divides both kings either vertically or horizontally
okapproachedcsquare	our king approached 'critical square', see Figure 15.9; here this means that the Manhattan distance has decreased
lpatt	'L-pattern' (Figure 15.9)
roomgt2	the 'room' for their king is greater than two squares

Move-constraints predicates

depth = N	move occurring at **depth = N** in the search tree
legal	any legal move
checkmove	checking move
rookmove	a rook move
nomove	fails for any move
kingdiagfirst	a king move, with preference for diagonal king moves

---

```
% King and rook vs. king in Advice Language 0

% Rules

edge_rule : if    their_king_edge and kings_close
            then [ mate_in_2, squeeze, approach, keeproom,
                   divide_in_2, divide_in_3 ].
```

```
else_rule : if    true
            then  [ squeeze, approach, keeproom,
                    divide_in_2, divide_in_3 ].

% Pieces-of-advice

advice( mate_in_2,
        mate :
        not rooklost and their_king_edge :
        (depth = 0) and legal then (depth = 2) and checkmove :
        (depth = 1) and legal).

advice( squeeze,
        newroomsmaller and not rookexposed and
        rookdivides and not stalemate :
        not rooklost :
        (depth = 0) and rookmove :
        nomove).

advice( approach,
        okapproachedcsquare and not rookexposed and
        (rookdivides or lpatt) and (roomgt2 or not our_king_edge) :
        not rooklost :
        (depth = 0) and kingdiagfirst :
        nomove).

advice( keeproom,
        themtomove and not rookexposed and rookdivides and okorndle and
        (roomgt2 or not okedge) :
        not rooklost :
        (depth = 0) and kingdiagfirst :
        nomove).

advice( divide_in_2,
        themtomove and rookdivides and not rookexposed :
        not rooklost :
        (depth < 3) and legal :
        (depth < 2) and legal).

advice( divide_in_3,
        themtomove and rookdivides and not rookexposed :
        not rooklost :
        (depth < 5) and legal :
        (depth < 4) and legal).
```

Figure 15.7 An AL0 advice-table for king and rook vs. king. The table consists of two rules and six pieces-of-advice.

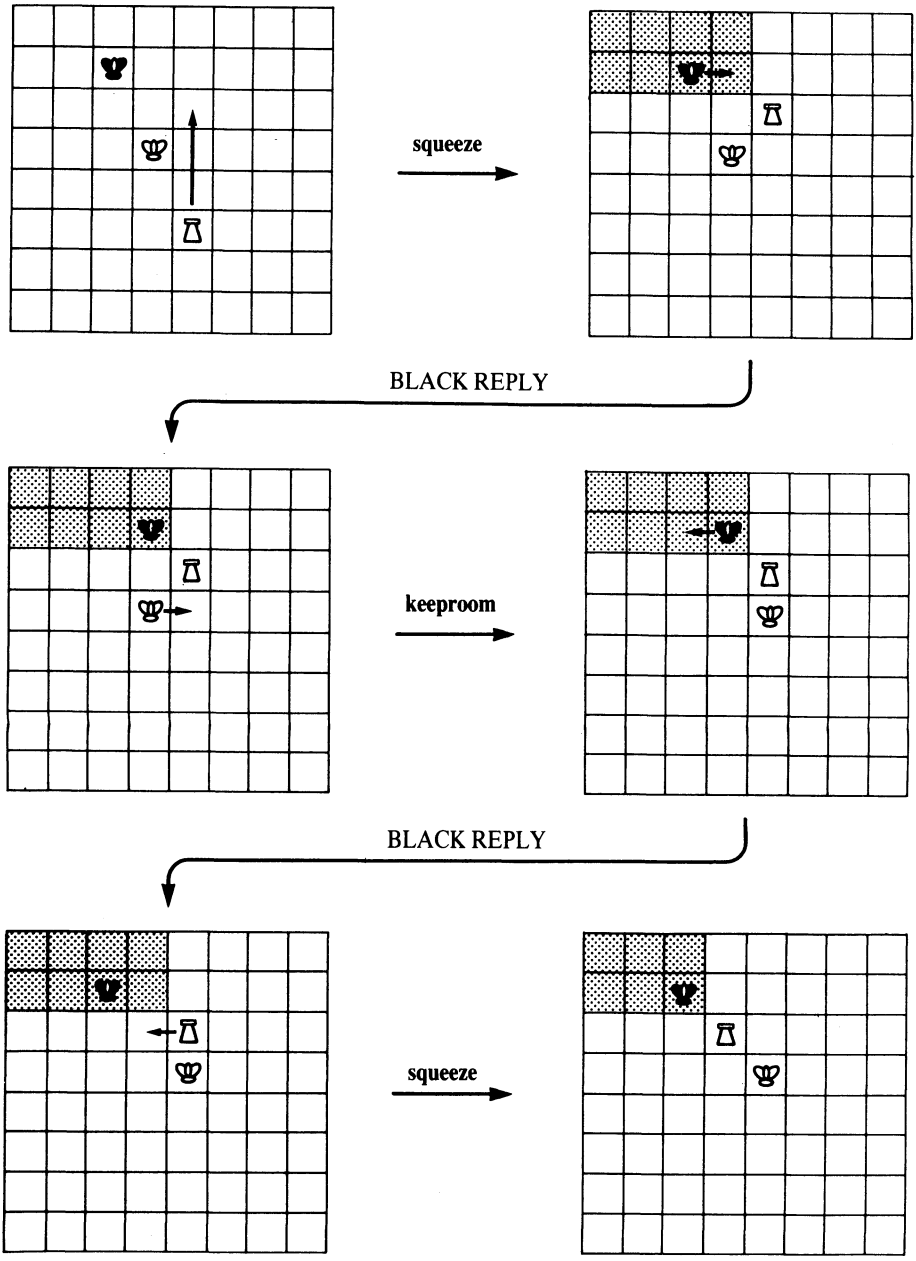


Figure 15.8 A game fragment played by the advice-table of Figure 15.7, illustrating the method of squeezing their king toward a corner. Pieces-of-advice used in this sequence are **keeproom** (waiting move preserving 'room') and **squeeze** ('room' has shrunk). The area to which their king is confined by our rook ('room') is shadowed. After the last **squeeze**, 'room' shrinks from eight to six squares.

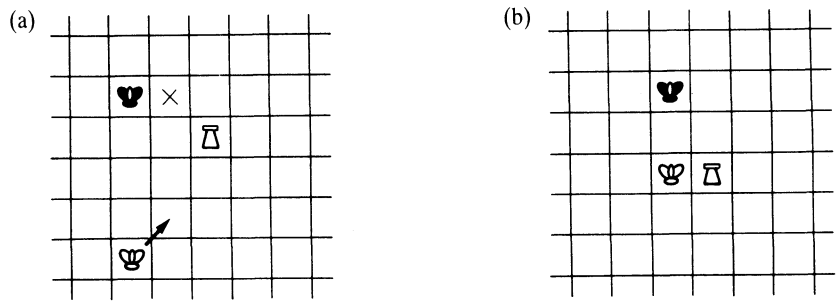


Figure 15.9 (a) Illustration of the 'critical square' (a crucial square in the squeezing manoeuvres, indicated by a cross); the White king approaches the critical square by moving as indicated. (b) The three pieces form an L-shaped pattern.

The arguments of these predicates are either positions (goal predicates) or moves (move-constraints predicates). Goal predicates can have one or two arguments. One argument is always the current search node; the second argument (if it exists) is the root node of the search tree. The second argument is needed in the so-called comparison predicates, which compare in some respect the root position and the current search position. An example is the predicate **newroomsmaller** which tests whether the 'room' for their king has shrunk (Figure 15.8). These predicates, together with chess rules for king and rook vs. king, and a board displaying procedure **(show(Pos))**, are programmed in Figure 15.10.

An example of how this advice-program plays is shown in Figure 15.8. The game would continue from the last position of Figure 15.8 as in the following variation (assuming 'them' moves as given in the variation). The algebraic chess notation is used where the files of the chessboard are numbered 'a', 'b', 'c', etc, and ranks are numbered 1, 2, 3, etc. For example, the move 'BK b7' means: move the Black king to the square in file 'b' and rank 7.

...	BK b7
WK d5	BK c7
WK c5	BK b7
WR c6	BK a7
WR b6	BK a8
WK b5	BK a7
WK c6	BK a8
WK **c7**	BK a7
WR c6	BK a8
WR a6	mate

Some questions can now be asked. First, is this advice-program *correct* in the sense that it mates against any defence if the game starts from any king and rook vs. king position? It is shown in Bratko (1978) by means of a formal proof

that an advice-table, effectively the same as the one in Figure 15.7, is correct in this sense.

Another question can be: Is this advice program *optimal* in the sense that it always delivers mate in the smallest number of moves? It can easily be shown by examples that the program's play is not optimal in this sense. It is known that optimal variations (optimally played by both sides) in this ending are at most 16 moves long. Although our advice-table can be rather far from this optimum, it was shown that the number of moves needed by this advice-table is still very safely under 50. This is important because of the 50-moves rule in chess: in endgames such as king and rook vs. king the stronger side has to mate within 50 moves; if not, a draw can be claimed.

```
% Predicate library for king and rook vs. king

% Position is represented by: Side..Wx : Wy..Rx : Ry..Bx : By..Depth
% Side is side to move ('w' or 'b')
% Wx, Wy are X and Y-coordinates of White king
% Rx, Ry are X and Y-coordinates of White rook
% Bx, By are coordinates of Black king
% Depth is depth of position in search tree

% Selector relations

side( Side..._, Side).
wk( _..WK..._, WK).
wr( _..._..WR..._, WR).
bk( _..._..._..BK..._, BK).
depth( _..._..._..._..Depth, Depth).

resetdepth( S..W..R..B..D, S..W..R..B..0).    % Copy of position with depth 0

% Some relations between squares

n( N, N1) :-                                  % Neighbour integers 'within board'
  ( N1 is N + 1;
    N1 is N - 1),
  in( N1).

in( N) :-
  N > 0, N < 9.

diagngb( X : Y, X1 : Y1) :-                   % Diagonal neighbour squares
  n( X, X1), n( Y, Y1).

verngb( X : Y, X : Y1) :-                     % Vertical neighbour squares
  n( Y, Y1).

horngb( X : Y, X1 : Y) :-                     % Horizontal neighbour squares
  n( X, X1).
```

```
ngb( S, S1) :-                          % Neighbour squares, first diagonal
  diagngb( S, S1);
  horngb( S, S1);
  verngb( S, S1).

end_of_game( Pos) :-
  mate( Pos).



% Move-constraints predicates
% These are specialized move generators:
% move( MoveConstr, Pos, Move, NewPos)

move( depth < Max, Pos, Move, Pos1) :-
  depth( Pos, D),
  D < Max, !.

move( depth = D, Pos, Move, Pos1) :-
  depth( Pos, D), !.

move( kingdiagfirst, w..W..R..B..D, W-W1, b..W1..R..B..D1) :-
  D1 is D + 1,
  ngb( W, W1),                          % 'ngb' generates diagonal moves first
  not ngb( W1, B),                      % Must not move into check
  W1 \== R.                             % Must not collide with rook

move( rookmove, w..W..Rx : Ry..B..D, Rx : Ry-R, b..W..R..B..D1) :-
  D1 is D + 1,
  coord( I),                            % Integer between 1 and 8
  ( R = Rx : I; R = I : Ry),            % Move vertically or horizontally
  R \== Rx : Ry,                        % Must have moved
  not inway( Rx : Ry, W, R).            % White king not in way

move( checkmove, Pos, R-Rx : Ry, Pos1) :-
  wr( Pos, R),
  bk( Pos, Bx : By),
  (Rx = Bx; Ry = By),                   % Rook and Black king in line
  move( rookmove, Pos, R-Rx : Ry, Pos1).

move( legal, w..P, M, P1) :-
  ( MC = kingdiagfirst; MC = rookmove),
  move( MC, w..P, M, P1).

move( legal, b..W..R..B..D, B-B1, w..W..R..B1..D1) :-
  D1 is D + 1,
  ngb( B, B1),
  not check( w..W..R..B1..D1).

legalmove( Pos, Move, Pos1) :-
  move( legal, Pos, Move, Pos1).
```

```
check( _..W..Rx : Ry..Bx : By..._) :-
  ngb( W, Bx : By);                          % King's too close
  ( Rx = Bx; Ry = By),
  Rx : Ry \== Bx : By,                       % Not rook captured
  not inway( Rx : Ry, W, Bx : By).

inway( S, S1, S1) :- !.

inway( X1 : Y, X2 : Y, X3 : Y) :-
  ordered( X1, X2, X3), !.

inway( X : Y1, X : Y2, X : Y3) :-
  ordered( Y1, Y2, Y3).

ordered( N1, N2, N3) :-
  N1 < N2, N2 < N3;
  N3 < N2, N2 < N1.

coord(1). coord(2). coord(3). coord(4).
coord(5). coord(6). coord(7). coord(8).


% Goal predicates

true( Pos).

themtomove( b..._).                          % Black = 'them' to move

mate( Pos) :-
  side( Pos, b),
  check( Pos),
  not legalmove( Pos, _, _).

stalemate( Pos) :-
  side( Pos, b),
  not check( Pos),
  not legalmove( Pos, _, _).

newroomsmaller( Pos, RootPos) :-
  room( Pos, Room),
  room( RootPos, RootRoom),
  Room < RootRoom.

rookexposed( Side..W..R..B..._) :-
  dist( W, R, D1),
  dist( B, R, D2),
  ( Side = w, !, D1 > D2 + 1;
    Side = b, !, D1 > D2).

okapproachedcsquare( Pos, RootPos) :-
  okcsquaremdist( Pos, D1),
  okcsquaremdist( RootPos, D2),
  D1 < D2.
```

```
okcsquaremdist( Pos, Mdist) :-   % Manh. dist. between WK and critical square
  wk( Pos, WK),
  cs( Pos, CS),                  % Critical square
  manhdist( WK, CS, Mdist).

rookdivides( _..Wx : Wy..Rx : Ry..Bx : By..._) :-
  ordered( Wx, Rx, Bx), !;
  ordered( Wy, Ry, By).

lpatt( _..W..R..B..._) :-                    % L-pattern
  manhdist( W, B, 2),
  manhdist( R, B, 3).

okorndle( _..W..R..._, _..W1..R1..._) :-
  dist( W, R, D),
  dist( W1, R1, D1),
  D =< D1.

roomgt2( Pos) :-
  room( Pos, Room),
  Room > 2.


our_king_edge( _..X : Y..._) :-              % White king on edge
  ( X = 1, !; X = 8, !; Y = 1, !; Y = 8).

their_king_edge( _..W..R..X : Y..._) :-      % Black king on edge
  ( X = 1, !; X = 8, !; Y = 1, !; Y = 8).

kings_close( Pos) :-                         % Distance between kings < 4
  wk( Pos, WK), bk( Pos, BK),
  dist( WK, BK, D),
  D < 4.

rooklost( _..W..B..B..._).                   % Rook has been captured

rooklost( b..W..R..B..._) :-
  ngb( B, R),                                % Black king attacks rook
  not ngb( W, R).                            % White king does not defend

dist( X : Y, X1 : Y1, D) :-                  % Distance in king moves
  absdiff( X, X1, Dx),
  absdiff( Y, Y1, Dy),
  max( Dx, Dy, D).

absdiff( A, B, D) :-
  A > B, !, D is A-B;
  D is B-A.

max( A, B, M) :-
  A >= B, !, M = A;
  M = B.
```

```
manhdist( X : Y, X1 : Y1, D) :-                    % Manhattan distance
  absdiff( X, X1, Dx),
  absdiff( Y, Y1, Dy),
  D is Dx + Dy.

room( Pos, Room) :-                                % Area to which B. king is confined
  wr( Pos, Rx : Ry),
  bk( Pos, Bx : By),
  ( Bx < Rx, SideX is Rx - 1; Bx > Rx, SideX is 8 - Rx),
  ( By < Ry, SideY is Ry - 1; By > Ry, SideY is 8 - Ry),
  Room is SideX * SideY, !;
  Room is 64.                                      % Rook in line with Black king

cs( _..W..Rx : Ry..Bx : By..._, Cx : Cy) :-        % 'Critical square'
  ( Bx < Rx, !, Cx is Rx - 1; Cx is Rx + 1),
  ( By < Ry, !, Cy is Ry - 1; Cy is Ry + 1).


% Display procedures

show( Pos) :-
  nl,
  coord( Y), nl,
  coord( X),
  writepiece( X : Y, Pos),
  fail.

show( Pos) :-
  side( Pos, S), depth( Pos, D),
  nl, write( 'Side= '), write( S),
  write( 'Depth= '), write( D), nl.

writepiece( Square, Pos) :-
  wk( Pos, Square), !, write( 'W');
  wr( Pos, Square), !, write( 'R');
  bk( Pos, Square), !, write( 'B');
  write( '.').

showmove( Move) :-
  nl, write( Move), nl.
```

Figure 15.10 Predicate library for king and rook vs. king.

Project

Consider some other simple chess endgame, such as king and pawn vs. king, and write an AL0 program (together with the corresponding predicate definitions) to play this endgame.

Summary

- Two-person games fit the formalism of AND/OR graphs. AND/OR search procedures can be therefore used to search game trees.
- The straightforward depth-first search of game trees is easy to program, but is too inefficient for playing interesting games. In such cases, the minimax principle, in association with an evaluation function and depth-limited search, offers a more feasible approach.
- The alpha-beta algorithm is an efficient implementation of the minimax principle. The efficiency of alpha-beta depends on the order in which alternatives are searched. In the best case, alpha-beta in effect reduces the branching factor of a game tree to its square root.
- Some refinements to the basic alpha-beta algorithm include: extending the search until a quiescent position is reached, progressive deepening and heuristic pruning.
- Numerical evaluation is a very restrictive form of applying game-specific knowledge. A more knowledge-intensive approach to game playing should provide for pattern-based knowledge. Advice Languages realize such an approach, where knowledge is represented in terms of goals and means of achieving goals.
- Programs written in this chapter are: an implementation of minimax and alpha-beta, an interpreter for Advice Language 0, and an advice-table for playing the king and rook vs. king chess endgame.
- Concepts introduced and discussed in this chapter are:

  two-person, perfect-information games
  game trees
  evaluation function, minimax principle
  static values, backed-up values
  alpha-beta algorithm
  progressive deepening, heuristic pruning, quiescence heuristics
  horizon effect
  Advice Languages
  goals, constraints, piece-of-advice, advice-table

References

The minimax principle, implemented as the alpha-beta algorithm, is the most popularly used approach to game-playing programs, in particular to chess programs. The minimax principle was introduced by Shannon (1950). The development of the alpha-beta technique had a rather complicated history when several researchers independently discovered or implemented the method or at least part of it. This interesting history is described by Knuth and

Moore (1975) who also present a more compact formulation of the alpha-beta algorithm using the 'neg-max' principle instead of minimax, and give a mathematical analysis of its performance. The most comprehensive treatment of several minimax-based algorithms and their analyses is Pearl (1984). There is another interesting question regarding the minimax principle: knowing that the static evaluation is only reliable to some degree, will the minimax backed-up values be more reliable than the static values themselves? Pearl (1984) has also collected results of mathematical analyses that pertain to this question. Results on error propagation in minimax trees explain when and why the minimax look-ahead is beneficial.

The collection of papers Bramer (1983) covers in depth several aspects of computer game playing. Frey (**1983**) is a good collection of articles on computer chess. On-going research on chess is published in the Advances in Computer Chess series and in the ICCA journal.

The Advice Language approach to using pattern knowledge in chess was introduced by Michie, and further developed in Bratko and Michie (1980a, b), and Bratko (1982, 1984, 1985). The king and rook vs. king advice-program of this chapter is a slight modification of the advice-table that was mathematically proved correct in Bratko (1978). Van Emden (1982) has also programmed this advice-table in Prolog.

Other interesting experiments in knowledge-intensive approach to chess (as opposed to search-intensive approaches) include Berliner (1977), Pitrat (1977) and Wilkins (1980).

Advances in Computer Chess Series (M. R. B. Clarke, ed.). Edinburgh University Press (Vols. 1–2), Pergamon Press (Vol. 3).

Berliner, H. J. (1977) A representation and some mechanisms for a problem solving chess program. In: *Advances in Computer Chess 1* (M. R. B. Clarke, ed.). Edinburgh University Press.

Bramer, M. A. (1983, ed.) *Computer Game Playing: Theory and Practice*. Ellis Horwood and John Wiley.

Bratko, I. (1978) Proving correctness of strategies in the AL1 assertional language. *Information Processing Letters* **7**: 223–230.

Bratko, I. (1982) Knowledge-based problem solving in AL3. In: *Machine Intelligence 10* (J. Hayes, D. Michie, J. H. Pao, eds.). Ellis Horwood (an abbreviated version also appears in Bramer 1983).

Bratko, I. (1984) Advice and planning in chess end-games. In: *Artificial and Human Intelligence* (S. Amarel, A. Elithorn, R. Banerji, eds.). North-Holland.

Bratko, I. (1985) Symbolic derivation of chess patterns. In: *Progress in Artificial Intelligence* (L. Steels, J. A. Campbell, eds.). Ellis Horwood and John Wiley.

Bratko, I. and Michie, D. (1980a) A representation of pattern-knowledge in chess endgames. In: *Advances in Computer Chess 2* (M. R. B. Clarke, ed.). Edinburgh University Press.

Bratko, I. and Michie, D. (1980b) An advice program for a complex chess programming task. *Computer Journal* **23**: 353–359.

Frey, P. W. (1983, ed.) *Chess Skill in Man and Machine* (second edition). Springer-Verlag.

Knuth, D. E. and Moore, R. W. (1975) An analysis of alpha-beta pruning. *Artificial Intelligence* **6**: 293–326.

Pearl, J. (1984) *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley.

Pitrat, J. (1977) A chess combination program which uses plans. *Artificial Intelligence* **8**: 275–321.

Shannon, C. E. (1950) Programming a computer for playing chess. *Philosophical Magazine* **41**: 256–275.

van Emden, M. (1982) Chess end-game advice: a case study in computer utilisation of knowledge. In: *Machine Intelligence 10* (J. Hayes, D. Michie, J. H. Pao, eds.). Ellis Horwood.

Wilkins, D. E. (1980) Using patterns and plans in chess. *Artificial Intelligence* **14**: 165–203.

16 Pattern-directed Programming

In this chapter we will consider pattern-directed systems as a special approach to programming. Prolog itself can be viewed as a pattern-directed language. We will implement a small interpreter for simple pattern-directed programs and illustrate the flavour of pattern-directed programming with examples.

16.1 Pattern-directed architecture

16.1.1 Main concepts

By *pattern-directed systems* we here refer to an architecture for program systems. This architecture is better suited for certain types of problems than conventional systems organization. Among problems that naturally fit into the pattern-directed architecture are many Artificial Intelligence applications – for example, expert systems. The main difference between conventional systems and pattern-directed systems is in the mechanisms of invocation of program modules. In conventional organization, modules of the system call each other according to a fixed, explicitly predefined scheme. Each program module decides which module will be executed next by *explicitly* calling other modules. The corresponding flow of execution is sequential and deterministic.

In contrast to this, in pattern-directed organization the modules of the system are not directly called by other modules. Instead, they are 'called' by *patterns* that occur in their 'data environment'. Therefore such modules are called *pattern-directed modules*. A *pattern-directed program* is a collection of pattern-directed modules. Each module is defined by:

(1) a precondition pattern, and
(2) an action to be executed if the data environment matches the pattern.

The execution of program modules is triggered by patterns that occur in the system's environment. The data environment is usually called the *database*. We can imagine such a system as shown in Figure 16.1.

There are some notable observations about Figure 16.1. There is no hierarchy among modules, and there is no explicit indication about which module can invoke some other module. Modules communicate with the

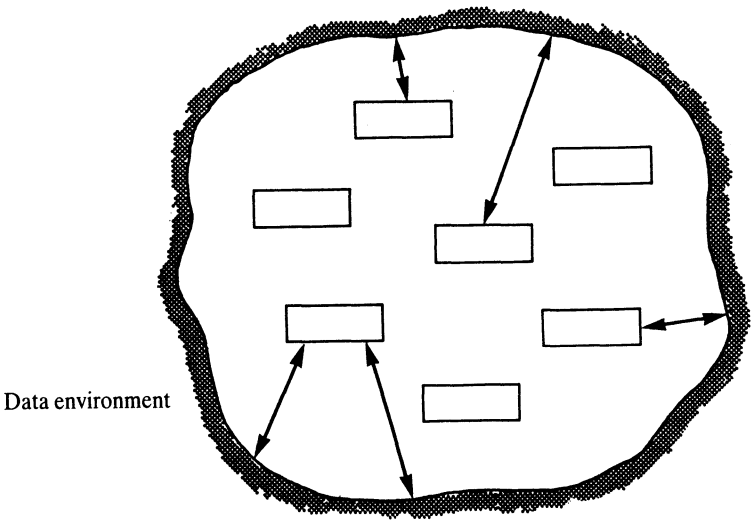


Figure 16.1 A pattern-directed system.

database rather than with other modules directly. The structure itself, in principle, permits execution of several modules in parallel, since the state of the database may simultaneously satisfy several preconditions and thus, in principle, fire several modules at the same time. Consequently such an organization can also serve as a natural model of parallel computation in which each module would be physically implemented by its own processor.

Pattern-directed architecture has certain advantages. One major advantage is that the design of the system does not require all the connections between modules to be carefully planned and defined in advance. Consequently, each module can be designed and implemented relatively autonomously. This renders a high degree of modularity. The modularity is manifested, for example, in that the removal of some module from the system is not necessarily fatal. After the removal, the system would often still be able to solve problems, only *the way* of solving problems might change. The same is true for the addition of new modules and for modifications of the existing modules. If similar modifications are carried out in systems with conventional organization, at least the calls between modules have to be properly modified.

The high degree of modularity is especially desirable in systems with complex knowledge bases because it is difficult to predict in advance all the interactions between individual pieces of knowledge in the base. The pattern-directed architecture offers a natural solution to this: each piece of knowledge, represented by an if-then rule, can be regarded as a pattern-directed module.

Let us further elaborate the basic scheme of pattern-directed systems with the view on an implementation. Figure 16.1 suggests that the parallel implementation would be most natural. However, let us assume the system is to be implemented on a traditional sequential processor. Then in a case that the

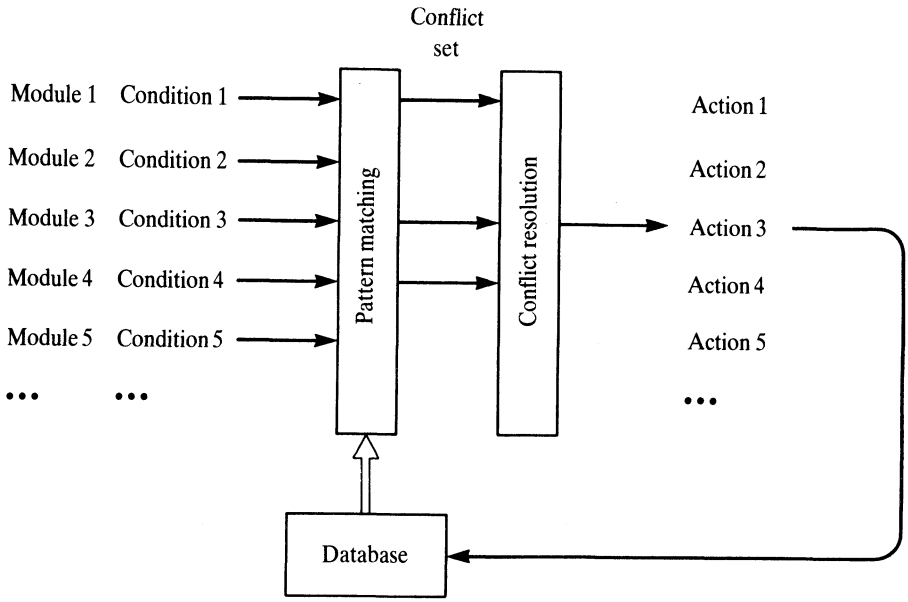


Figure 16.2 The basic life cycle of pattern-directed systems. In this example the database satisfies the condition pattern of modules 1, 3 and 4; module 3 is chosen for execution.

triggering patterns of several modules simultaneously occur in the database there is a conflict: which of all these potentially active modules will actually be executed? The set of potentially active modules is called a *conflict set*. In an actual implementation of the scheme of Figure 16.1 on a sequential processor, we need an additional program module, called the *control module*. The control module resolves the conflict by choosing and activating one of the modules in the conflict set. One simple rule of resolving conflicts can be based on a predefined, fixed ordering of modules.

The basic life cycle of pattern-directed systems, then, consists of three steps:

(1) *Pattern matching*: find in the database all the occurrences of the condition patterns of the program modules. This results in a conflict set.

(2) *Conflict resolution*: choose one of the modules in the conflict set.

(3) *Execution*: execute the module that was chosen in step 2.

This implementational scheme is illustrated in Figure 16.2.

16.1.2 Prolog programs as pattern-directed systems

Prolog programs themselves can be viewed as pattern-directed systems. Without much elaboration, the correspondence between Prolog and pattern-

directed systems is along the following lines:

- Each Prolog clause in the program can be viewed as a pattern-directed module. The module's condition part is the head of the clause, the action part is specified by the clause's body.
- The system's database is the current list of goals that Prolog is trying to satisfy.
- A clause is fired if its head matches the first goal in the database.
- To execute a module's action (body of a clause) means: replace the first goal in the database with the list of goals in the body of the clause (with the proper instantiation of variables).
- The process of module invocation is non-deterministic in the sense that several clauses' heads may match the first goal in the database, and any one of them can, in principle, be executed. This non-determinism is actually implemented in Prolog through backtracking.

16.1.3 Writing pattern-directed programs: an example

Pattern-directed systems can also be viewed as a particular style of writing programs and thinking about problems, called *pattern-directed programming*.

To illustrate this, consider an elementary programming exercise: computing the greatest common divisor D of two integer numbers A and B. The classical Euclid's algorithm can be written as follows:

> To compute the greatest common divisor, D, of A and B:
>
> While A and B are not equal, repeat the following:
>
> if $A > B$ then replace A with $A - B$
> else replace B with $B - A$.
>
> When this loop is over, A and B are equal; now the greatest common divisor D is A (or B).

We can define the same process by two pattern-directed modules:

Module 1

Condition	There are two numbers X and Y in the database such that $X > Y$.
Action	Replace X in the database with the difference $X - Y$.

Module 2

Condition	There is a number X in the database.
Action	Output X and stop.

Obviously, whenever the condition of Module 1 is satisfied, so is the condition of Module 2 and we have a conflict. This will be in our case resolved by a simple

control rule: Module 1 is always preferred to Module 2. Initially the database contains the two numbers A and B.

As a pleasant surprise, our pattern-directed program in fact solves a more general problem: computing the greatest common divisor of any number of integers. If several integers are stored in the database the system will output the greatest common divisor of all of them. Figure 16.3 shows a possible sequence of changes in the database before the result is obtained, when the initial database contains four numbers: 25, 10, 15, 30. Notice that a module's precondition can be satisfied at several places in the database.

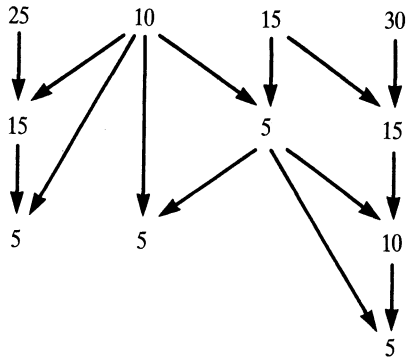


Figure 16.3 A possible execution of the pattern-directed program for computing the greatest common divisor of a set of numbers. In this example the database initially contains the numbers 25, 10, 15 and 30. Vertical arrows connect numbers with their replacements. The final state of the database is: 5, 5, 5, 5.

We will in this chapter implement an interpreter for a simple language for specifying pattern-directed systems, and illustrate the flavour of pattern-directed programming by programming exercises.

16.2 A simple interpreter for pattern-directed programs

Let us choose the following syntax for specifying pattern-directed modules:

ConditionPart ---> ActionPart

The condition part is a list of conditions

[Cond1, Cond2, Cond3, ...]

where **Cond1**, **Cond2**, etc. are simply Prolog goals. The precondition is

satisfied if all the goals in the list are satisfied. The action part is a list of actions:

```
[ Action1, Action2, ...]
```

Each action is, again, simply a Prolog goal. To execute an action list, all the actions in the list have to be executed. That is, all the corresponding goals have to be satisfied. Among available actions there will be actions that manipulate the database: *add*, *delete* or *replace* objects in the database.

Figure 16.4 shows our pattern-directed program for computing the greatest common divisor written in this syntax.

The simplest way to implement this pattern-directed language is to use Prolog's own built-in database mechanism. Adding an object into the database and deleting an object can be accomplished simply by the built-in procedures:

```
assert( Object)          retract( Object)
```

Replacing an object with another object is also easy:

```
replace( Object1, Object2) :-
  retract( Object1), !,
  assert( Object2).
```

The cut in this clause is used just to prevent **retract** from deleting (through backtracking) more than one object from the database.

```
% Production rules for finding greatest common divisor ( Euclid algorithm)

:- op( 300, fx, number).

[ number X, number Y, X > Y] --->
[ NewX is X - Y, replace( number X, number NewX) ].

[ number X]  --->  [ write( X), stop].

% An initial database

number 25.

number 10.

number 15.

number 30.
```

Figure 16.4 A pattern-directed program to find the greatest common divisor of a set of numbers.

```
% A small interpreter for pattern-directed programs
% The system's database is manipulated through assert/retract

:- op( 800, xfx, --->).

run :-
  Condition ---> Action,          % A rule
  test( Condition),               % Precondition satisfied?
  execute( Action).

test( [] ).                       % Empty condition

test( [First | Rest] ) :-         % Test conjunctive condition
  call( First),
  test( Rest).

execute( [stop] ) :- !.           % Stop execution

execute( [] ) :-                  % Empty action (execution cycle completed)
  run.                            % Continue with next execution cycle

execute( [ First | Rest] ) :-
  call( First),
  execute( Rest).

replace( A, B) :-                 % Replace A with B in database
  retract( A), !,
  assert( B).
```

Figure 16.5 A small interpreter for pattern-directed programs.

A small interpreter for pattern-directed programs along these lines is shown in Figure 16.5. This interpreter is perhaps an oversimplification in some respects. In particular, the conflict resolution rule in the interpreter is extremely simple and rigid: always execute the *first* potentially active pattern-directed module (in the order as they are written). So the programmer's control is reduced just to the ordering of modules. The initial state of the database for this interpreter has to be asserted as Prolog clauses. Then the execution is triggered by the goal:

```
?- run.
```

16.3 A simple theorem prover

Let us implement a simple theorem prover as a pattern-directed system. The prover will be based on the *resolution principle*, a popular method for mechani-

cal theorem proving. We will limit our discussion to only proving theorems in the simple *propositional logic* just to illustrate the principle, although our resolution mechanism will be easily extendable to handle the first-order predicate calculus (logic formulas that contain variables). Basic Prolog itself is a special case of such a theorem prover.

The theorem-proving task can be defined as: given a formula, show that the formula is a theorem; that is, the formula is always true regardless of the interpretation of the symbols that occur in the formula. For example, the formula

$$p \text{ v } \sim p$$

read as 'p or not p', is always true regardless of the meaning of p.

We will be using the following symbols as logic operators:

~	negation, read as 'not'
&	conjunction, read as 'and'
v	disjunction, read as 'or'
=>	implication, read as 'implies'

The precedence of these operators is such that 'not' binds strongest, then 'and', then 'or', and then 'implies'.

In the resolution method we negate the conjectured theorem and then try to show that this negated formula is a contradiction. If the negated formula is in fact a contradiction then the original formula must be a tautology. Thus the idea is: demonstrating that the negated formula is a contradiction is equivalent to proving that the original formula is a theorem (always holds). The process that aims at detecting the contradiction consists of a sequence of *resolution steps*.

Let us illustrate the principle with a simple example. Suppose we want to prove that the following propositional formula is a theorem:

$$(a \Rightarrow b) \ \& \ (b \Rightarrow c) \Rightarrow (a \Rightarrow c)$$

This formula is read as: if b follows from a, and c follows from b, then c follows from a.

Before the resolution process can start we have to get our negated, conjectured theorem into a form that suits the resolution process. The suitable form is the *conjunctive normal form*, which looks like this:

$$(p_1 \text{ v } p_2 \text{ v } \ldots) \ \& \ (q_1 \text{ v } q_2 \text{ v } \ldots) \ \& \ (r_1 \text{ v } r_2 \text{ v } \ldots) \ \& \ \ldots$$

Here all p's, q's and r's are simple propositions or their negations. This form is also called the *clause form*. Each conjunct is called a *clause*. So $(p_1 \text{ v } p_2 \text{ v } \ldots)$ is a clause.

We can easily transform any propositional formula into this form. For our

example theorem, this transformation can proceed as follows. The theorem is

$$(a => b) \& (b => c) => (a => c)$$

The negated theorem is:

$$\sim ((a => b) \& (b => c) => (a => c))$$

The following known equivalence rules will be useful when transforming this formula into the normal conjunctive form:

(1)	$x => y$	is equivalent to	$\sim x \text{ v } y$
(2)	$\sim(x \text{ v } y)$	is equivalent to	$\sim x \& \sim y$
(3)	$\sim(x \& y)$	is equivalent to	$\sim x \text{ v } \sim y$
(4)	$\sim(\sim x)$	is equivalent to	x

Applying rule 1 to our formula we get:

$$\sim (\sim ((a => b) \& (b => c)) \text{ v } (a => c))$$

By rules 2 and 4 we get:

$$(a => b) \& (b => c) \& \sim(a => c)$$

Using rule 1 at several places we get:

$$(\sim a \text{ v } b) \& (\sim b \text{ v } c) \& \sim(\sim a \text{ v } c)$$

By rule 2 we finally get the clause form we need:

$$(\sim a \text{ v } b) \& (\sim b \text{ v } c) \& a \& \sim c$$

This consists of four clauses. Now the resolution process can start.

The basic resolution step can occur any time that there are two clauses such that some proposition p occurs in one of them, and $\sim p$ occurs in the other. Let two such clauses be:

$$p \text{ v } Y \quad \text{and} \quad \sim p \text{ v } Z$$

where p is a proposition, and Y and Z are propositional formulas. Then the resolution step on these two clauses produces a third clause:

$$Y \text{ v } Z$$

It can easily be shown that this clause logically follows from the two initial clauses. So by adding the expression (Y v Z) to our formula we do not alter the

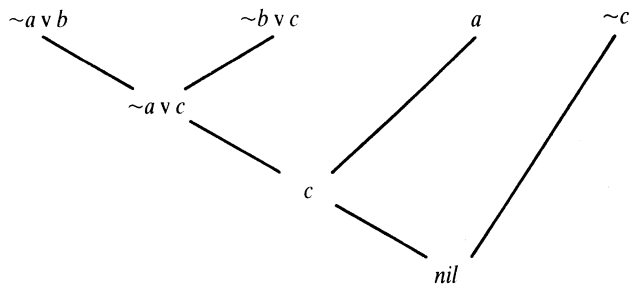


Figure 16.6 Proving the theorem $(a => b)$ & $(b => c) => (a => c)$ by the resolution method. The top line is the negated theorem in the clause form. The empty clause at the bottom signals that the negated theorem is a contradiction.

validity of the formula. The resolution process thus generates new clauses. If the 'empty clause' (usually denoted by 'nil') occurs then this will signal that a contradiction has been found. The empty clause *nil* is generated from two clauses of the forms:

x and $\sim x$

which is obviously a contradiction.

Figure 16.6 shows the resolution process that starts with our negated conjectured theorem and ends with the empty clause.

Figure 16.7 shows how this resolution process can be formulated as a pattern-directed program. This program operates on clauses asserted into the database. The resolution principle can be formulated as a pattern-driven activity:

if

there are two clauses *C1* and *C2*, such that *P* is a (disjunctive) subexpression of *C1*, and $\sim P$ is a subexpression of *C2*

then

remove *P* from *C1* (giving *CA*), remove $\sim P$ from *C2* (giving *CB*), and add into the database a new clause: *CA* v *CB*.

Written in our pattern-directed language this becomes:

```
[ clause( C1), delete( P, C1, CA),
  clause( C2), delete( ~P, C2, CB) ] --->
[ assert( clause( CA v CB) ) ].
```

This rule needs a little elaboration to prevent repeated actions on the same clauses, which would merely produce new copies of already existing clauses. The program in Figure 16.7 records into the database what has already been done by asserting:

```
done( C1, C2, P)
```

The condition parts of rules will then recognize and prevent such repeated actions.

The rules in Figure 16.7 also deal with some special cases that would otherwise require the explicit representation of the empty clause. Also, there are two rules that just simplify clauses when possible. One of these rules removes redundant subexpressions. For example, this rule would simplify the clause

$a \vee b \vee a$

into $a \vee b$. The other rule recognizes true clauses such as

$a \vee b \vee \sim a$

and removes them from the database since they are useless for detecting a contradiction.

A remaining question is how to translate a given propositional formula into the clause form. This is not difficult and the program of Figure 16.8 does it. The procedure

translate(Formula)

translates a formula into a set of clauses C1, C2, etc., and asserts these clauses into the database as:

```
clause( C1).
clause( C2).
...
```

Now the pattern-directed theorem prover can be triggered by the goal **run**. So, to prove a conjectured theorem using these programs, we translate the negated theorem into the clause form and start the resolution process. For our example theorem, this is done by the question:

```
?- translate( ~( ( a => b) & ( b => c) => ( a => c) ) ), run.
```

The program will respond with 'Contradiction found' meaning that the original formula is a theorem.

```
% Production rules for resolution theorem proving

% Contradicting clauses

[ clause( X), clause( ~X) ] --->
[ write( 'Contradiction found'), stop].
```

```
% Remove a true clause

[ clause( C), in( P, C), in( ~P, C) ] --->
[ retract( C) ].

% Simplify a clause

[ clause( C), delete( P, C, C1), in( P, C1) ] --->
[ replace( clause( C), clause( C1) ) ].

% Resolution step, a special case

[ clause( P), clause( C), delete( ~P, C, C1), not done( P, C, P) ] --->
[ assert( clause( C1) ), assert( done( P, C, P) ) ].

% Resolution step, a special case

[ clause( ~P), clause( C), delete( P, C, C1), not done( ~P, C, P) ] --->
[ assert( clause( C1) ), assert( done( ~P, C, P) ) ].

% Resolution step, general case

[ clause( C1), delete( P, C1, CA),
  clause( C2), delete( ~P, C2, CB), not done( C1, C2, P) ] --->
[ assert( clause( CA v CB) ), assert( done( C1, C2, P) ) ].

% Last rule: resolution process stuck

[] ---> [ write( 'Not contradiction'), stop].

% delete( P, E, E1) means: delete a disjunctive subexpression P from E giving E1

delete( X, X v Y, Y).

delete( X, Y v X, Y).

delete( X, Y v Z, Y v Z1) :-
  delete( X, Z, Z1).

delete( X, Y v Z, Y1 v Z) :-
  delete( X, Y, Y1).

% in( P, E) means: P is a disjunctive subexpression in E

in( X, X).

in( X, Y) :-
  delete( X, Y, _).
```

Figure 16.7 A pattern-directed program for simple resolution theorem proving.

```
% Translating a propositional formula into (asserted) clauses

:- op( 100, fy, ~).                          % Negation
:- op( 110, xfy, &).                         % Conjunction
:- op( 120, xfy, v).                         % Disjunction
:- op( 130, xfy, =>).                        % Implication

translate( F & G) :- !,                      % Translate conjunctive formula
  translate( F),
  translate( G).

translate( Formula) :-
  transform( Formula, NewFormula), !,        % Transformation step on Formula
  translate( NewFormula).

translate( Formula) :-                       % No more transformation possible
  assert( clause( Formula) ).

% Transformation rules for propositional formulas

transform( ~( ~X), X) :- !.                  % Double negation

transform( X >= Y, ~X v Y) :- !.             % Eliminate implication

transform( ~( X & Y), ~X v ~Y) :- !.         % De Morgan's law

transform( ~( X v Y), ~X & ~Y) :- !.         % De Morgan's law

transform( X & Y v Z, ( X v Z) & ( Y v Z) ) :- !. % Distribution

transform( X v Y & Z, ( X v Y) & ( X v Z) ) :- !. % Distribution

transform( X v Y, X1 v Y) :-
  transform( X, X1), !.                      % Transform subexpression

transform( X v Y, X v Y1) :-
  transform( Y, Y1), !.                      % Transform subexpression

transform( ~X, ~X1) :-
  transform( X, X1).                         % Transform subexpression
```

Figure 16.8 Translating a propositional calculus formula into a set of (asserted) clauses.

16.4 Concluding remarks

Our simple interpeter for pattern-directed programs was sufficient for illustrating some ideas of pattern-directed programming. For more complex applications it should be elaborated in several respects. Here are some critical comments and indications for improvements.

The conflict resolution was in our interpreter reduced to a fixed, predefined order. Much more flexible schemas are often desired. To enable more sophisticated control, all the potentially active modules should be found and fed into a special user-programmable control module.

When the database is large and there are many pattern-directed modules in the program then pattern matching can become extremely inefficient. The efficiency in this respect can be improved by a more sophisticated organization of the database. This may involve the indexing of the information in the database, or partition of the information into sub-bases, or partition of the set of pattern-directed modules into subsets. The idea of partitioning is to make only a *subset* of the database or of the modules accessible at any given time, thus reducing the pattern matching to such a subset only. Of course, in such a case we would need a more sophisticated control mechanism that would control the transitions between these subsets in the sense of activating and de-activating a subset. A kind of meta-rules could be used for that.

Unfortunately our interpreter, as programmed, precludes any backtracking due to the way that the database is manipulated through **assert** and **retract**. So we cannot study alternative execution paths. This can be improved by using a different implementation of the database, avoiding Prolog's **assert** and **retract**. One way would be to represent the whole state of the database by a Prolog term passed as an argument to the **run** procedure. The simplest possibility is to organize this term as a list of objects in the database. The interpreter's top level could then look like this:

```
run( State) :-
  Condition ---> Action,
  test( Condition, State),
  execute( Action, State).
```

The **execute** procedure would then compute a new state and call **run** with this new state.

Project

Implement an interpreter for pattern-directed programs that does not maintain its database as Prolog's own internal database (with **assert** and **retract**), but as a procedure argument according to the foregoing remark. Such a new interpreter would allow for automatic backtracking. Try to design a representation of the database that would facilitate efficient pattern matching.

Summary

- Pattern-directed architecture suits many problems of Artificial Intelligence.

- A pattern-directed program is a collection of pattern-directed modules whose execution is triggered by patterns in the 'database'.
- Prolog programs themselves can be viewed as pattern-directed systems.
- The parallel implementation of pattern-directed systems would be most natural. The sequential implementation requires *conflict resolution* among the modules in the *conflict* set.
- A simple interpreter for pattern-directed programs was implemented in this chapter and applied to resolution-based theorem proving in propositional logic.
- Concepts discussed in this chapter are:

  pattern-directed systems, pattern-directed architecture
  pattern-directed programming
  pattern-directed module
  conflict set, conflict resolution
  resolution-based theorem proving, resolution principle

References

Waterman and Hayes-Roth (1978) is the classical book on pattern-directed systems. Fundamentals of mechanical theorem proving can be found in Nilsson (1980), including the algorithm for transforming the first-order predicate calculus formulas into the clausal form. Clocksin and Mellish (1981) give a Prolog program that does this transformation.

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

Nilsson, N. J. (1980) *Principles of Artificial Intelligence*. Tioga; also Springer-Verlag.

Waterman, D. A. and Hayes-Roth, F. (1978, eds.) *Pattern-Directed Inference Systems*. Academic Press.

Solutions to Selected Exercises

Chapter 1

1.1 (a) no
(b) X = pat
(c) X = bob
(d) X = bob, Y = pat

1.2 (a) `?- parent( X, pat).`
(b) `?- parent( liz, X).`
(c) `?- parent( Y, pat), parent( X, Y).`

1.3 (a)
```
happy( X) :-
  parent( X, Y).
```
(b)
```
hastwochildren( X) :-
  parent( X, Y),
  sister( Z, Y).
```

1.4
```
grandchild( X, Z) :-
  parent( Y, X),
  parent( Z, Y).
```

1.5
```
aunt( X, Y) :-
  parent( Z, Y),
  sister( X, Z).
```

1.6 Yes it is.

1.7 (a) no backtracking
(b) no backtracking
(c) no backtracking
(d) backtracking

Chapter 2

2.1 (a) variable
(b) atom

(c) atom
(d) variable
(e) atom
(f) structure
(g) number
(h) syntactically incorrect
(i) structure
(j) structure

2.3 (a) yes
(b) no
(c) no
(d) **D = 2, E = 2**
(e) **P1 = point(–1,0)**
P2 = point(1,0)
P3 = point(0,Y)
This can represent the family of triangles with two vertices on the x-axis at 1 and –1 respectively, and the third vertex anywhere on the y-axis.

2.4
```
seg( point(5,Y1), point(5,Y2) )
```

2.5
```
regular( rectangle( point(X1,Y1), point(X2,Y1), point(X2,Y3),
point(X1,Y3) ) ).
```
% This assumes that the first point is the left bottom vertex

2.6 (a) **A = 2**
(b) **no**
(c) **C = 7**
(d) **D = s(s(1));**
D = s(s(s(s(1))))

2.7
```
relatives( X, Y) :-
  predecessor( X, Y);
  predecessor( Y, X);
  predecessor( Z, X),
  predecessor( Z, Y);
  predecessor( X, Z),
  predecessor( Y, Z).
```

2.8
```
translate( 1, one).
translate( 2, two).
translate( 3, three).
```

2.9 In the case of Figure 2.10 Prolog does slightly more work.

2.10 According to the definition of matching of Section 2.2, this succeeds. X becomes a sort of circular structure in which X itself occurs as one of the arguments.

Chapter 3

3.1 (a)
```
conc( L1, [_, _, _], L)
```

(b)
```
conc( [_, _, _], L1, L),          % Delete first three elements from L
conc( L2, [_, _, _], L1)          % Delete last three from L1
```

3.2 (a)
```
last( Item, List) :-
  conc( _, [Item], List).
```

(b)
```
last( Item, [Item] ).

last( Item, [First | Rest] ) :-
  last( Item, Rest).
```

3.3
```
evenlength( [] ).

evenlength( [First | Rest] ) :-
  oddlength( Rest).

oddlength( [_] ).

oddlength( [First | Rest] ) :-
  evenlength( Rest).
```

3.4
```
reverse( [], [] ).

reverse( [First | Rest], Reversed) :-
  reverse( Rest, ReversedRest),
  conc( ReversedRest, [First], Reversed).
```

3.5
```
% This is easy using reverse
palindrome( List) :-
  reverse( List, List).

% Alternative solution, not using reverse
palindrome1( [] ).

palindrome1( [_] ).

palindrome1( [First | Rest] ) :-
  conc( Middle, [First], Rest),
  palindrome1( Middle).
```

3.6
```
shift( [First | Rest], Shifted) :-
  conc( Rest, [First], Shifted).
```

3.7
```
translate( [], [] ).

translate( [Head | Tail], [Head1 | Tail1] ) :-
  means( Head, Head1),
  translate( Tail, Tail1).
```

3.8
```
subset( [], [] ).

subset( [First | Rest], [First | Sub] ) :-          % Retain First in subset
  subset( Rest, Sub).

subset( [First | Rest], Sub) :-                     % Remove First
  subset( Rest, Sub).
```

3.9
```
dividelist( [], [], [] ).                          % Nothing to divide

dividelist( [X], [X], [] ).                        % Divide one-element list

dividelist( [X, Y | List], [X | List1], [Y | List2] ) :-
  dividelist( List, List1, List2).
```

3.10
```
canget( state( _, _, _, has), [] ).                % Nothing to do

canget( State, [Action | Actions] ) :-
  move( State, Action, NewState),                  % First action
  canget( NewState, Actions).                      % Remaining actions
```

3.11
```
flatten( [Head | Tail], FlatList) :-               % Flatten non-empty list
  flatten( Head, FlatHead),
  flatten( Tail, FlatTail),
  conc( FlatHead, FlatTail, FlatList).

flatten( [], [] ).                                 % Flatten empty list

flatten( X, [X] ).                                 % Flatten a non-list

% Note: On backtracking this program produces rubbish
```

3.12
```
Term1 = plays( jimmy, and( football, squash) )
Term2 = plays( susan, and( tennis, and( basketball, volleyball) ) )
```

3.13
```
:- op( 300, xfx, was).
:- op( 200, xfx, of).
:- op( 100, fx, the).
```

3.14
```
(a)  A = 1+0
(b)  B = 1+1+0
(c)  C = 1+1+1+1+0
(d)  D = 1+1+0+1
```

3.15
```
:- op( 100, xfx, in).
:- op( 300, fx, concatenating).
:- op( 200, xfx, gives).
:- op( 100, xfx, and).
:- op( 300, fx, deleting).
:- op( 100, xfx, from).
```

```
% List membership
Item in [Item | List].

Item in [First | Rest] :-
  Item in Rest.

% List concatenation
concatenating [] and List gives List.

concatenating [X | L1] and L2 gives [X | L3] :-
  concatenating L1 and L2 gives L3.

% Deleting from a list
deleting Item from [Item | Rest] gives Rest.

deleting Item from [First | Rest] gives [First | NewRest] :-
  deleting Item from Rest gives NewRest.
```

```
3.16 max( X, Y, X) :-
       X >= Y.

     max( X, Y, Y) :-
       X < Y.
```

```
3.17 maxlist( [X], X).                  % Maximum of single-element list

     maxlist( [X, Y | Rest], Max),      % At least two elements in list
       maxlist( [Y | Rest], MaxRest),
       max( X, MaxRest, Max).           % Max is the greater of X and MaxRest
```

```
3.18 sumlist( [], 0).

     sumlist( [First | Rest], Sum) :-
       sumlist( Rest, SumRest),
       Sum is First + SumRest.
```

```
3.19 ordered( [X] ).                    % Single-element list is ordered

     ordered( [X, Y | Rest] ) :-
       X =< Y,
       ordered( [Y | Rest] ).
```

```
3.20 subsum( [], 0, [] ).

     subsum( [N | List], Sum, [N | Sub] ) :-      % N is in subset
       Sum1 is Sum - N,
       subsum( List, Sum1, Sub).

     subsum( [N | List], Sum, Sub) :-             % N is not in subset
       subsum( List, Sum, Sub).
```

```
3.21 between( N1, N2, N1) :-
       N1 =< N2.
```

```
between( N1, N2, X) :-
  N1 < N2,
  NewN1 is N1 + 1,
  between( NewN1, N2, X).
```

3.22
```
:- op( 900, fx, if).
:- op( 800, xfx, then).
:- op( 700, xfx, else).
:- op( 600, xfx, :=).

if Val1 > Val2 then Var := Val3 else Anything :-
  Val1 > Val2,
  Var = Val3.

if Val1 > Val2 then Anything else Var := Val4 :-
  Val1 =< Val2,
  Var = Val4.
```

Chapter 4

4.1 (a)
```
?- family( person( _, Name, _, _), _, [] ).
```

(b)
```
?- child( person( Name, SecondName, _, works( _, _) ) ).
```

(c)
```
?- family( person( _, Name, _, unemployed),
           person( _, _, _, works( _, _) ) ).
```

(d)
```
?- family( Husband, Wife, Children),
   dateofbirth( Husband, date( _, _, Year1) ),
   dateofbirth( Wife, date( _, _, Year2) ),
   ( Year1 - Year2 >= 15;
     Year2 - Year1 >= 15 ),
   member( Child, Children).
```

4.2
```
twins( Child1, Child2) :-
  family( _, _, Children),
  del( Child1, Children, OtherChildren),        % Delete Child1
  member( Child2, OtherChildren),
  dateofbirth( Child1, Date),
  dateofbirth( Child2, Date).
```

4.3
```
nth_member( 1, [X | L], X).          % X is first element of list [X | L]

nth_member( N, [Y | L], X) :-        % X is nth element of [Y | L]
  N1 is N - 1,
  nth_member( N1, L, X).
```

4.4 The input string shrinks on each non-silent cycle, and it cannot shrink indefinitely.

4.5

```
accepts( S, [], _) :-
  final( S).

accepts( S, [X | Rest], MaxMoves) :-
  MaxMoves > 0,
  trans( S, X, S1),
  NewMax is MaxMoves – 1,
  accepts( S1, Rest).

accepts( S, String, MaxMoves) :-
  MaxMoves > 0,
  silent( S, S1),
  NewMax is MaxMoves – 1,
  accepts( S1, String, NewMax).
```

4.7 (a)

```
jump( X/Y, X1/Y1) :-          % Knight jump from X/Y to X1/Y1
  ( dxy( Dx, Dy);             % Knight distances in x and y directions
    dxy( Dy, Dx) ),           % or the other way round
  X1 is X + Dx,
  inboard( X1),               % X1 is within chessboard
  Y1 is Y + Dy,
  inboard( Y1).               % Y1 is within chessboard

dxy( 2, 1).                   % 2 squares to right, 1 forward
dxy( 2, –1).                  % 2 squares to right, 1 backward
dxy( –2, 1).                  % 2 to left, 1 forward
dxy( –2, –1).                 % 2 to left, 1 backward

inboard( Coord) :-            % Coordinate within chessboard
  0 < Coord,
  Coord < 9.
```

(b)

```
knightpath( [Square] ).       % Knight sitting on Square

knightpath( [S1, S2 | Rest] ) :-
  jump( S1, S2),
  knightpath( [S2 | Rest] ).
```

(c)

```
?- knightpath( [2/1,R,5/4,S,X/8] ).
```

Chapter 5

5.1 (a)

```
X = 1;
X = 2;
```

(b)

```
X = 1
Y = 1;

X = 1
Y = 2;
```

```
X = 2
Y = 1;

X = 2
Y = 2;
```

(c)
```
X = 1
Y = 1;

X = 1
Y = 2;
```

5.2
```
class( Number, positive) :-
  Number > 0, !.

class( 0, zero) :- !.

class( Number, negative).
```

5.3
```
split( [], [], [] ).

split( [X | L], [X | L1], L2) :-
  X >= 0, !,
  split( L, L1, L2).

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

5.4
```
member( Item, Candidates), not member( Item, RuledOut)
```

5.5
```
difference( [], _, [] ).

difference( [X | L1], L2, L) :-
  member( X, L2), !,
  difference( L1, L2, L).

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

5.6
```
unifiable( [], _, [] ).

unifiable( [First | Rest], Term, List) :-
  not( First = Term), !,
  unifiable( Rest, Term, List).

unifiable( [First | Rest], Term, [First | List] ) :-
  unifiable( Rest, Term, List).
```

Chapter 6

6.1
```
findterm( Term) :-
  read( Term), !,                % Assuming current input stream is file f
  write( Term);                  % Current term in  f matches Term?
  findterm( Term).               % If yes, display it
                                 % Otherwise process the rest of file
```

6.2
```
findallterms( Term) :-
  read( CurrentTerm),
  process( CurrentTerm, Term).

process( end_of_file, _) :- !.

process( CurrentTerm, Term) :-
  ( not( CurrentTerm = Term), !;     % Terms do not match
    write( CurrentTerm), nl),        % Otherwise output current term
  findallterms( Term).               % Do the rest of file
```

6.4
```
starts( Atom, Character) :-
  name( Character, [Code] ),
  name( Atom, [Code | _] ).
```

6.5
```
plural( Noun, Nouns) :-
  name( Noun, CodeList),
  name( s, CodeS),
  conc( CodeList, CodeS, NewCodeList),
  name( Nouns, NewCodeList).
```

Chapter 7

7.1
```
sum( X, Y, Sum) :-
  var( Sum), !,                  % Sum uninstantiated
  Sum is X + Y;
  var( X), !,                    % X uninstantiated
  X is Sum – Y;
  Y is Sum – X.                  % Y uninstantiated
```

7.2
```
add( Item, List) :-
  var( List), !,                 % List represents empty list
  List = [Item | Tail].

add( Item, [_ | Tail] ) :-
  add( Item, Tail).

member( X, List) :-
  var( List), !,                 % List represents empty list
  fail.                          % so X cannot be a member

member( X, [X | Tail] ).

member( X, [_ | Tail] ) :-
  member( X, Tail).
```

Chapter 8

8.2
```
add_at_end( L1 - [Item | Z2], Item, L1 - Z2).
```

8.3
```
reverse( A - Z, L - L) :-                    % Result is empty list if
  A == Z, !.                                 % A - Z represents empty list

reverse( [X | L] - Z, RL - RZ) :-            % Non-empty list
  reverse( L - Z, RL - [X | RZ] ).
```

Chapter 9

9.1
```
list( [] ).

list( [_ | Tail] ) :-
  list( Tail).
```

9.2
```
mem( X, X then Anything).

mem( X, Y then List) :-
  mem( X, List).
```

9.3
```
convert( [], donothing).

convert( [First | Tail], First then Rest) :-
  convert( Tail, Rest).
```

9.4
```
convert( [], EmptyList, _, EmptyList).       % Convert empty list

convert( [First | Tail], NewList, Functor, Empty) :-
  NewList =.. [Functor, First, NewTail],
  convert( Tail, NewTail, Functor, Empty).
```

9.8
```
msort( [], [] ).

msort( [X], [X] ).

msort( List, SortedList) :-
  divide( List, List1, List2),               % Divide into approx. equal lists
  msort( List1, Sorted1),
  msort( List2, Sorted2),
  merge( Sorted1, Sorted2, Sorted).          % Merge sorted lists

divide( [], [], [] ).


divide( [X], [X], [] ).

divide( [X, Y | L], [X | L1], [Y | L2] ) :-  % Put X, Y into separate lists
  divide( L, L1, L2).
```

9.9 (a)
```
binarytree( nil).

binarytree( t( Left, Root, Right) ) :-
  binarytree( Left),
  binarytree( Right).
```

9.10
```
height( nil, 0).

height( t( Left, Root, Right), H) :-
  height( Left, LH),
  height( Right, RH),
  max( LH, RH, MH),
  H is 1 + MH.

max( A, B, A) :-
  A >= B, !.

max( A, B, B).
```

9.11
```
linearize( nil, [] ).

linearize( t( Left, Root, Right), List) :-
  linearize( Left, List1),
  linearize( Right, List2),
  conc( List1, [Root | List2], List).
```

9.12
```
maxelement( t( _, Root, nil), Root) :- !.      % Root is right-most element

maxelement( t( _, _, Right), Max) :-           % Right subtree non-empty
  maxelement( Right, Max).
```

9.13
```
in( Item, t( _, Item, _), [Item] ).

in( Item, t( Left, Root, _), [Root | Path] ) :-
  gt( Root, Item),
  in( Item, Left, Path).

in( Item, t( _, Root, Right), [Root | Path] ) :-
  gt( Item, Root),
  in( Item, Right, Path).
```

9.14
```
% Display a binary tree from top to bottom
% This program assumes that each node is just one character

show( Tree) :-
  dolevels( Tree, 0, more).                % Do all levels from top

dolevels( Tree, Level, alldone) :- !.      % No more nodes beyond level

dolevels( Tree, Level, more) :-            % Do all levels from level
  traverse( Tree, Level, 0, Continue), nl, % Output nodes at Level
  NextLevel is Level + 1,
  dolevels( Tree, NextLevel, Continue).    % Do lower levels
```

```
traverse( nil, _, _, _).

traverse( t( Left, X, Right), Level, Xdepth, Continue) :-
  NextDepth is Xdepth + 1,
  traverse( Left, Level, NextDepth, Continue),      % Traverse left subtree
  ( Level = Xdepth, !,                              % Node X at Level?
    write( X), Continue = more;                     % Output node, more to do
    write(' ') ),                                   % Otherwise leave space
  traverse( Right, Level, NextDepth, Continue).     % Traverse right subtree
```

Chapter 10

10.1

```
in( Item, l( Item) ).                   % Item found in leaf

in( Item, n2( T1, M, T2) ) :-           % Node has two subtrees
  gt( M, Item), !,                      % Item not in second subtree
  in( Item, T1);                        % Search first subtree
  in( Item, T2).                        % Otherwise search the second

in( Item, n3( T1, M2, T2, M3, T3) ) :-  % Node has three subtrees
  gt( M2, Item), !,                     % Item not in second or third
  in( Item, T1);                        % Search first subtree
  gt( M3, Item), !,                     % Item not in third subtree
  in( Item, T2);                        % Search second subtree
  in( Item, T3).                        % Search third subtree
```

10.3

```
avl( Tree) :-
  avl( Tree, Height).                   % Tree is AVL-tree with height Height

avl( nil, 0).                           % Empty tree is AVL and has height 0

avl( t( Left, Root, Right), H) :-
  avl( Left, HL),
  avl( Right, HR),
  ( HL is HR; HL is HR + 1; HL is HR - 1),   % Subtrees heights almost equal
  max1( HL, HR, H).

max1( U, V, M) :-                       % M is 1 + max of U and V
  U > V, !, M is U + 1;
  M is V + 1.
```

Chapter 11

11.1

```
depthfirst1( [Node | Path], [Node | Path] ) :-
  goal( Node).

depthfirst1( [Node | Path], Solution) :-
  s( Node, Node1),
  not member( Node1, Path),
  depthfirst1( [Node1, Node | Path], Solution).
```

11.6
```
solve( StartSet, Solution) :-          % StartSet is list of start nodes
bagof( [Node], member( Node, StartSet), CandidatePaths),
breadthfirst( CandidatePaths, Solution).
```

Index

2-3 dictionary 234
  deleting from 239
  insertion into 235
2-3 tree 234
=.. 163
== 169
\== 169

admissibility of search
  AND/OR 305
  state space 273
Advice Language 0 372
  implementation of 375
  interpreter for 374
Advice Languages 370, 372
advice-table 372
alpha-beta algorithm 364
  efficiency of 367
  program for 366
AL0 372
  implementation of 375
AND node 288
AND/OR best-first search 299, 306
AND/OR breadth-first search 298
AND/OR depth-first search 298
AND/OR formulation of game
  playing 293
AND/OR formulation of route
  finding 290, 310
AND/OR graph 286, 288
  search admissibility 305
AND/OR search procedures 294
AND/OR solution tree 289
anonymous variable 30
arg 167
arithmetic in Prolog 84
arithmetic operators 85
arity 33
assert 170
asserta 172
assertz 172
atom 156
atom 7
atomic 156
atoms
  constructing 149
  decomposing 149
  syntax of 28
automatic backtracking 24, 47
automaton, non-deterministic 99
AVL-dictionary 241
  insertion into 241
AVL-tree 241

backed-up values 362
bactracking
  automatic 24, 47
  controlling 120
  preventing 120
bagof 175
balanced tree 216
best-first search 265, 266
  AND/OR 299, 306
  state space 266
better-goal 370
bidirectional search 262
binary dictionary 214
  deletion 219
  insertion 217, 220
  searching 214
binary tree 211
body of clause 10
breadth-first search 256
bubblesort 206
built-in operators
  +, –, *, /, **div**, **is**, **mod** 85
built-in procedures
  >, <, >=, =<, =:=, =\= 86
  ! 121, 124
  =.. 163
  == 169
  \== 169
  arg 167
  assert 170
  asserta 172
  assertz 172

atom 156
atomic 156
bagof 175
call 166
consult 152
findall 176
functor 166
get 148
get0 148
integer 155
is 85
name 149
nl 140
nonvar 156
nospy 188
not 131
notrace 187
numbervars 341
put 147
read 140
reconsult 153
repeat 174
retract 170
see 138
seen 139
setof 176
spy 187
tab 140
tell 138
told 139
trace 187
ttyflush 142
var 156
write 140

call 166
categorical knowledge 317
certainty factor 347
certainty propagation 350
chess, king-rook-king ending 374
clause 4
  body of 10, 13
  head of 10, 13
  instance of 41
  variant of 41
clause form 61, 397
clause in propositional logic 397
closed-world assumption 135
combinatorial complexity 263
combinatorial explosion 263
comments in Prolog 19
comparison operators 86
compiled program 188
concatenation of lists 68
conflict resolution 392
conflict set 392
conjunction of goals 42
conjunctive normal form 397
consult 152
cryptarithmetic puzzles, program for 161
cut 120, 124
  green 134
  red 134

data abstraction 97
database manipulation 169
database, Prolog 169
debugging 187
declarative meaning 24, 42
declarative reading 41
depth-first search 251
depth-first search, depth-limited 255
dictionary
  2-3 234
  AVL 241
  binary 214
difference list 192
directed graph 224
disjunction of goals 42
div operator 85

eight puzzle 247
eight queens problem 108
end_of_file 139, 140
equality, types of 168
evaluation function in games 368
executing goals 43
expert system 314
expert system shell 315
expert systems, uncertainty in 347

facts in Prolog 13
file **user** 138
files in Prolog 137
files of terms 140
findall 176
first-order predicate logic 61
forcing-tree 371
functor 166
functor 31
  arity of 33
  principal 33

game tree 359
game playing, AND/OR formulation of 293
games, two-person, perfect-information 359

generalization, use of 182
get 148
get0 148
goal 7
  to satisfy 19
  conjunction of 42
  disjunction of 42
goal is satisfiable 8
goal is unsatisfiable 8
goal logically follows 20
goal succeeds 8
goals
  executing 43
  executing list of 46
graph
  AND/OR 286, 288
  directed 224
  path finding 225
graphs, representation of 224

Hamiltonian path 227
head of clause 10
head of list 64
heuristic estimates 265
heuristic pruning 369
heuristic search 263, 265
holding-goal 370
horizon effect 369
Horn clause 61
how explanation 328, 342, 356
how question 317, 328

if-then rules 316
inference engine 315
inference network 348
insertion sort 207
instance of clause 41
instantiation, most general 36
instantiation of variable 10
integer 155, 156
interpreting program 188
is operator 85

king-rook-king ending, program for 378
knowledge base 315
knowledge-based system 314
knowledge engineering 317

list
  adding to 72, 126
  concatenation 68
  decomposition of 69
  deleting from 72
  difference-pair representation 192
  head of 64
  inserting into 73
  length of 88
  membership 67, 125
  permutation of 75
  representation of 203
  sorting of 206
  sublist of 73
  tail of 64
list structure 64
lists in Prolog 64
logic
  first-order predicate 61
  propositional 397
  relation to Prolog 60

main variation 362
map colouring, program for 189
matching 21, 35, 36
meaning, procedural 24, 43
meaning of Prolog
  declarative 24, 42
  procedural 24, 43
measure of belief 347
memoization of answers 173
minimax-based programs 368
minimax principle 361
minimax procedure 363
mod operator 85
most general instantiation 36
move-constraints 370

name 149
necessity factor 349
negation as failure 129
nl 140
non-deterministic automaton 99
nonvar 156
nospy 188
not 131
notrace 187
numbers in Prolog 29
numbervars 341

objects in Prolog 27
operator notation 78
operators
  comparison 86
  precedence of 79
  predefined 82
  type of 80
operators in Prolog 78
OR node 288

path finding 224
pattern-directed module 390
pattern-directed program 390
pattern-directed programming 390, 393
pattern-directed programs, interpreter for 394
pattern-directed systems 390
pattern-directed systems, conflict resolution in 392
pattern-directed systems, pattern matching in 392
permutation of list 75
piece-of-advice 370
  satisfiability of 371
precedence of operators 79
predicate logic 61
principal functor 33
probabilistic knowledge 317
problem reduction 286
procedural meaning 24, 43
procedural reading 41
procedure in Prolog 19
procedures, tabular organization of 186
programming style 179
  commenting 186
  rules of 184
programming techniques
  debugging 187
  efficiency 188
progressive deepening 369
Prolog, relation to logic 60
propositional logic 397
Prospector 347
Prospector model 347
pure Prolog 61
put 147

quicksort 208

read 140
reconsult 153
recursion 17
  use of 181
recursive definition 17
repeat 174
resolution principle 61, 396
resolution step 398
resolution theorem proving 61
  program for 401
retract 170
route finding, AND/OR formulation of 290, 310
rules, if-then 316
rules in Prolog 9, 13

scheduling 278
see 138
seen 139
selector relations 97
setof 176
shell, expert system 315
soft knowledge 317
sorting lists 206
spanning tree, program for 229, 230
spy 187
state space 247
  representation 247, 248
  search admissibility 273
static values 362
stepwise refinement 180
stream
  input 137
  output 137
streams 137
structured objects 30
structured programming 181
structures in Prolog 30
subjective certainty 347
sufficiency factor 349

tab 140
tail of list 64
task scheduling 278
tell 138
term 31
  copy of 174, 177
  ground 167
  subsumption of 168
theorem prover 396
told 139
tower of Hanoi program 292
trace 187
transparency of system 317
tree
  2-3 234
  AVL 241
  balanced 216
  binary 211
  displaying 222
ttyflush 142

uncertainty in expert systems 347
unification 61
user file 138

var 156
variable 7
  anonymous 30
  instantiation of 10
variables, syntax of 29
variant of clause 41

why explanation 327
why question 317, 327
write 140



INTERNATIONAL
COMPUTER SCIENCE
SERIES

Prolog – the logic programming language central to Fifth Generation computers – is now recognized as a powerful vehicle for non-numeric programming. This book provides a detailed account of the language and its applications in Artificial Intelligence.

Part One introduces Prolog as a practical programming tool, and shows how Prolog programs are developed. Part Two demonstrates the power of Prolog applied in some central areas of AI, including

- Problem solving and heuristic search
- Expert systems
- Game playing
- Pattern-directed systems

Fundamental AI techniques are developed in depth towards their implementation in Prolog, resulting in complete Prolog programs.

The author, Ivan Bratko, leads the AI groups at the Josef Stefan Institute and the E. Kardelj University in Ljubljana, Yugoslavia. He is also a visiting fellow of the Turing Institute, Glasgow, and a director of the International School for the Synthesis of Expert Knowledge. Professor Bratko has taught Prolog worldwide and applied Prolog in medical expert systems and computer chess research. This book will be of particular interest to students, researchers and professional programmers seeking to use Prolog for AI applications.



ADDISON-WESLEY
PUBLISHING
COMPANY

ISBN 0 201 14224 4