PART VI

LANGUAGES AND PROGRAMMING TECHNIQUES FOR ARTIFICIAL INTELLIGENCE

for now we see as through a glass darkly . . .

—PAUL TO THE CORINTHIANS

The map is not the territory; the name is not the thing named.

—ALFRED KORZYBSKI

What have I learned but the proper use of several tools?

—GARY SNYDER, "What Have I Learned"

Languages, Understanding, and Levels of Abstraction

In Part VI we first discuss the issues involved in selecting a language for artificial intelligence programming. Then, in the chapters dedicated to LISP and PROLOG, we introduce a number of programming techniques for use in building intelligent systems. The primary function of AI programming is to construct the representation and control structures needed for intelligent problem solving. The requirements of these structures to a great extent determine the features that an AI implementation language should provide. In the Introduction to Part VI, we first enumerate the language features desired for AI programming and then introduce the LISP and PROLOG programming languages. Not only are these two of the most frequently used languages in artificial intelligence: their syntactic and semantic features also encourage powerful ways of thinking about problems and their solutions. The remarkable influence these languages have had on the historical development of AI is as much a product of their ability to function as "tools for thinking" as it is a reflection of their strengths as programming languages.

The ability to form higher-level abstractions from the particulars of experience is one of the most powerful and fundamental abilities of the human mind. Abstraction allows us to consolidate the details of a complicated domain into a general characterization of its organization and behavior; these abstractions allow us to understand the full range of particulars found in that domain. If we enter a strange house, for example, we will be able to find our way around: the organization of the living room, bedrooms, kitchen, and bathrooms generally conforms to a standard model of a house. The abstraction lets us make sense of the variations found in different houses. A picture may be worth a thousand words, but an abstraction can concisely represent the important features of an entire class of pictures.

When we form theories to describe classes of phenomena, the significant qualitative and quantitative features of the class are abstracted out from the details that characterize its individual members. This loss of detail is compensated for by the descriptive and predictive power of a valid theory. Abstraction is an essential tool for understanding and managing the complexity of the world around us, as well as that of our own mental structures. Indeed, this process of abstraction occurs continuously and recursively in the act of knowing: knowledge is built in layers of abstraction, from the mechanisms that extract structure from the chaos of raw sensory stimuli all the way up to the most subtle of scientific theories. Ultimately, most of our ideas are about other ideas.

Hierarchical abstraction, the organization of experience into increasingly abstract classes and descriptions, is an essential tool for understanding the behavior and organization of complex systems, including computer programs. Just as the behavior of an animal may be studied without concern for the underlying physiology of its nervous system, an algorithm has a characterization of its own, quite separate from the program that implements it.

Consider, for example, two different implementations of binary search, one written in FORTRAN using arrays and calculations on array indices and the other written in C++ using pointers to implement binary search trees. In a deep sense, these programs are the same, even though the particulars of their implementations differ. This separation of an algorithm from the code used to implement it is only one example of hierarchical abstraction in computer science.

Allen Newell has distinguished between the *knowledge level* and the *symbol level* of describing an intelligent system (Newell 1982). The symbol level is concerned with the particular formalisms used to represent problem-solving knowledge; the discussion of predicate logic as a representation language in Chapter 2 is an example of such a symbol-level consideration. Above the symbol level is the knowledge level, concerned with the knowledge content of the program and the way in which that knowledge is used.

This distinction is reflected in the architecture of knowledge-based systems and the development style it supports. Because users understand programs in terms of their knowledge and capabilities, it is important that AI programs have a clear knowledge-level characterization. The separation of the knowledge base from the underlying control structure makes this point of view explicit and simplifies the development of coherent, knowledge-level behavior. Similarly, the symbol level defines a representation language, such as logic or production rules, for the knowledge base. Its separation from the knowledge level allows the programmer to address issues of expressiveness, efficiency

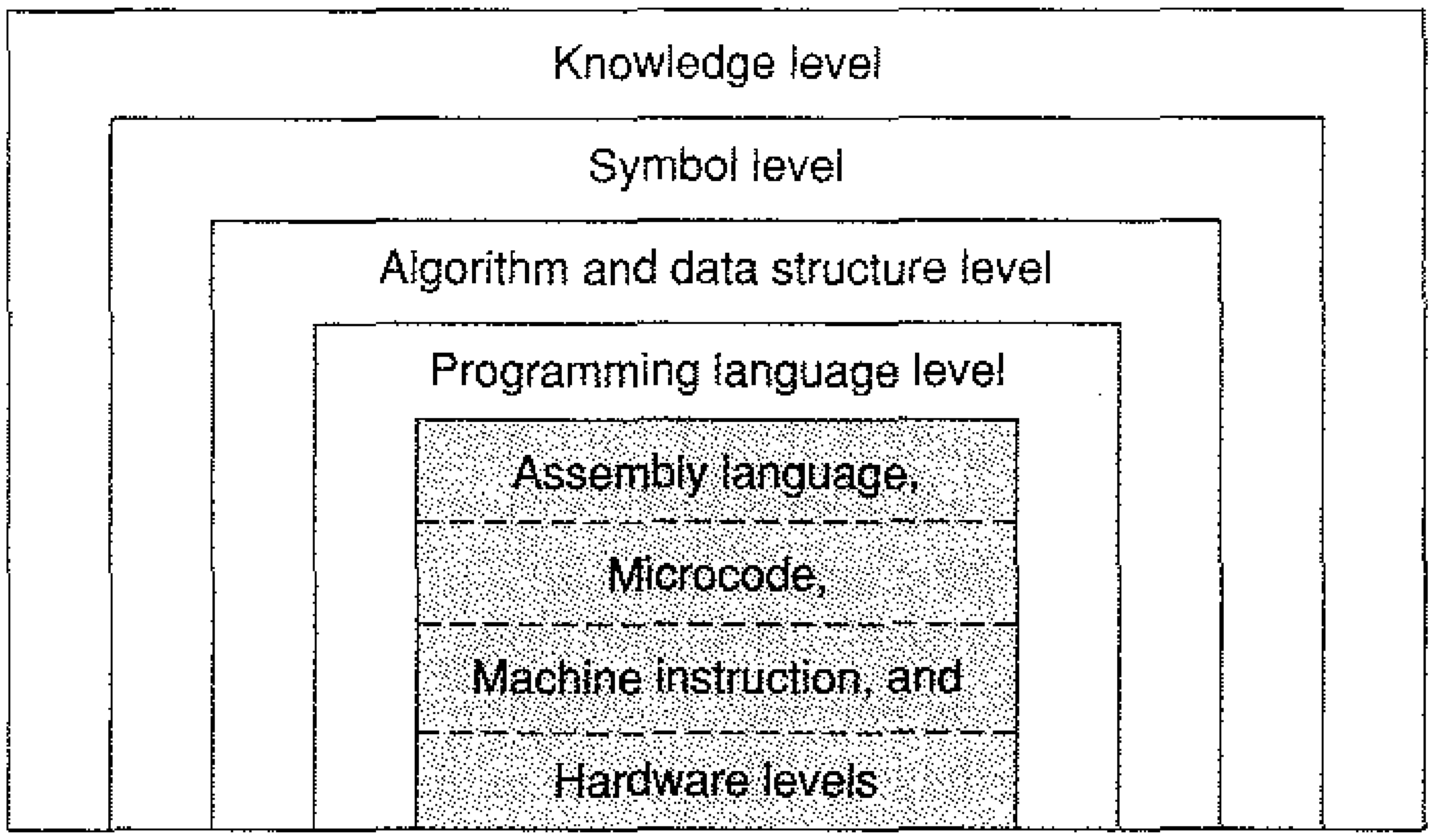

Figure VI.1 Levels of a knowledge-based system, adapted from Newell (1982).

and ease of programming that are not relevant to the program's higher-level behavior. The implementation of the symbol-level representation constitutes a still lower level of program organization and defines an additional set of design considerations, as in Figure VI.1.

The importance of the multi-level approach to system design cannot be overemphasized: it allows a programmer to ignore the complexity hidden at lower levels and focus on issues appropriate to the current level of abstraction. It allows the theoretical foundations of artificial intelligence to be kept free of the nuances of a particular implementation or programming language. It allows us to modify an implementation, improving its efficiency or porting it to another machine, without affecting its specification and behavior at higher levels.

The knowledge level defines the capabilities of an intelligent system. The knowledge content is independent of the formalisms used to represent it, as long as the representation language is sufficiently expressive. Knowledge-level concerns include such questions as: What queries will be made of the system? What objects and relations are important in the domain? How is new knowledge added to the system? Will facts change over time? How will the system need to reason about its knowledge? Does the domain of discourse have a well-understood taxonomy? Does the domain involve uncertain or missing information? Careful analysis at this level is an important step in designing the architecture of the program and in choosing the particular method of representation used at the symbol level.

At the symbol level, decisions are made about the structures used to represent and organize knowledge. The selection of a representation language is a primary symbol-level concern. As we have seen in Chapters 7, 8, and 9, logic is only one of many formalisms currently available for knowledge representation. Not only must a representation language be able to express the knowledge required for an application, but it also must be concise, modifiable, computationally efficient and must assist the programmer in acquiring and organizing the knowledge base. These goals often conflict and necessitate trade-offs in the design of representation languages.

Just as we have distinguished between the knowledge and symbol levels of a program, we can also distinguish between the symbol level and the algorithms and data structures used to implement it. For example, with the exception of efficiency, the behavior of a logic-based problem solver should be unaffected by the choice between a hash table and a binary tree to implement a table of its symbols. These are implementation decisions and should be invisible at the symbol level. Many of the algorithms and data structures used in implementing representation languages for AI are common computer science techniques such as binary trees and tables; others are more specific to AI and are presented in pseudo-code throughout the text and in the chapters on LISP and PROLOG.

Below the algorithm/data structure level is the language level. It is here that the implementation language for the program becomes significant. Even though good programming style requires that we build barriers of abstraction between the particular features of a programming language and the layers above it, the unique needs of symbol-level programming exert a profound influence on the design and use of AI programming languages. In addition, language design must accommodate the constraints it inherits from still lower levels of computer architecture, including the operating system, the underlying hardware architecture and the limitations physical computers must place on resources such as memory and processor speed. The techniques LISP and PROLOG use to mediate the needs of the symbol level and the requirements of the underlying architecture are both a source of their utility and also an intellectual achievement of great importance and elegance.

We next introduce the major AI programming languages, PROLOG and LISP.

An Overview of PROLOG and LISP

PROLOG

PROLOG is the best-known example of a *logic programming language*. A logic program is a set of specifications in formal logic; PROLOG uses the first-order predicate calculus. Indeed, the name itself comes from PROgramming in LOGic. An interpreter executes the program by systematically making inferences from logic specifications. The idea of using the representational power of the first-order predicate calculus to express specifications for problem solving is one of the central contributions PROLOG has made to computer science in general and to artificial intelligence in particular. The benefits of using first-order predicate calculus for a programming language include a clean and elegant syntax and well-defined semantics.

The implementation of PROLOG has its roots in research on theorem proving by J.A. Robinson (1965), especially the creation of algorithms for resolution refutation. Robinson designed a proof procedure called *resolution*, which is the primary method for computing with PROLOG. The chapter on automated theorem proving demonstrates *resolution refutation systems*; see Sections 13.2 and 13.3.

Because of these features, PROLOG has proved to be a useful vehicle for investigating such experimental programming issues as automatic code generation, program verification, and design of high-level specification languages. PROLOG and other logic-based languages support a declarative programming style—that is, constructing a program in terms of high-level descriptions of a problem's constraints—rather than a procedural programming style—writing programs as a sequence of instructions for performing an algorithm. This mode of programming essentially tells the computer "what is true" and "what needs to be done" rather than "how to do it." This allows programmers to focus on problem solving as sets of specifications for a domain rather than the details of writing low-level algorithmic instructions for "what to do next."

The first PROLOG program was written in Marseille, France, in the early 1970s as part of a project in natural language understanding (Colmerauer et al. 1973, Roussel 1975, Kowalski 1979*a*). The theoretical background for the language is discussed in the work of Kowalski, Hayes, and others (Kowalski 1979*a*, 1979*b*; Hayes 1977, Lloyd 1984). The major development of the PROLOG language was carried out from 1975 to 1979 at the department of artificial intelligence of the University of Edinburgh. The group in Edinburgh responsible for the implementation of PROLOG were David H.D. Warren and Fernando Pereira. They produced the first PROLOG interpreter robust enough for delivery to the general computing community. This product was built on the DEC-system 10 and could operate in both interpretive and compiled modes (Warren et al. 1979). Further descriptions of this early code and comparisons of PROLOG with LISP may be found in Warren et al. (1977). This "Warren and Pereira" PROLOG became the early standard, and the book *Programming in PROLOG* (Clocksin and Mellish 1984) was the chief vehicle for delivering PROLOG to the computing community. Our text uses this standard, which has come to be known as the Edinburgh syntax.

The advantages of the language have been demonstrated by research projects designed to evaluate and extend the expressive power of logic programming. Discussion of many such applications can be found in the Proceedings of the International Joint Conference on Artificial Intelligence and the Symposium on Logic Programming. See also the references at the end of Chapter 15.

LISP

LISP was first proposed by John McCarthy in the late 1950s. The language was originally intended as an alternative model of computation based on the theory of recursive functions. In an early paper, McCarthy (1960) outlined his goals: to create a language for symbolic rather than numeric computation, to implement a model of computation based on the theory of recursive functions (Church 1941), to provide a clear definition of the language's syntax and semantics, and to demonstrate formally the completeness of this computational model. Although LISP is one of the oldest computing languages still in existence (along with FORTRAN and COBOL), the careful thought given to its original design and the extensions made to the language through its history have kept it in the vanguard of programming languages. In fact, this programming model has proved so effective that a number of other languages have been based on functional programming, e.g., SCHEME, ML, and FP.

The list is the basis of both programs and data structures in LISP: LISP is an acronym for LISt Processing. LISP provides a powerful set of list-handling functions implemented internally as linked pointer structures. LISP gives programmers the full power and generality of linked data structures while freeing them from the responsibility for explicitly managing pointers and pointer operations.

Originally, LISP was a compact language, consisting of functions for constructing and accessing lists, defining new functions, detecting equality, and evaluating expressions. The only means of program control were recursion and a single conditional. More complicated functions, when needed, were defined in terms of these primitives. Through time, the best of these new functions became part of the language itself. This process of extending the language by adding new functions led to the development of numerous dialects of LISP, often including hundreds of specialized functions for data structuring, program control, real and integer arithmetic, input/output (I/O), editing LISP functions, and tracing program execution. These dialects are the vehicle by which LISP has evolved from a simple and elegant theoretical model of computing into a rich, powerful, and practical environment for building large software systems. Because of the proliferation of early LISP dialects, the Defense Advanced Research Projects Agency in 1983 proposed a standard dialect of the language, known as Common LISP.

Although Common LISP has emerged as the lingua franca of LISP dialects, a number of other dialects continue to be widely used. One of these is SCHEME, an elegant rethinking of LISP that has been used both for AI development and for teaching the fundamental concepts of computing. The dialect we use throughout our text is Common LISP.

Selecting an Implementation Language

As artificial intelligence has matured and demonstrated its applicability to a range of practical problems, its almost exclusive reliance on LISP and PROLOG has diminished. The circumstances of software development, such as the need to easily interface with legacy code, the use of AI as modules of large, conventional programs, and the need to conform to development standards imposed by corporate or government customers has led to the development of AI systems in a variety of languages, including Smalltalk, C, C++, and Java. Nonetheless, LISP and PROLOG continue to be important for prototyping and development and an important part of any AI programmer's skill set.

In addition, these languages have served as proving grounds for many of the features that continue to be incorporated into modern programming languages. Perhaps the best example of this is the Java language, which profits from its use of dynamic binding, automatic memory management, and other features that were pioneered in AI languages. It seems as though the rest of the programming world is still trying to catch up to the standards set by AI languages. As this evolution continues, knowledge of LISP, PROLOG, or Smalltalk and the programming techniques they enable will only increase in value. We are confident that this will be true, whether you continue to use one of these classic AI languages, or find yourself programming in C++, Objective C, Java or one of their other competitors, descendants, or distant cousins.

AN INTRODUCTION TO PROLOG

All the objects of human reason or inquiry may naturally be divided into two kinds, to wit, "Relations of Ideas" and "Matters of Fact."

—DAVID HUME, *An Inquiry Concerning Human Understanding*

The only way to rectify our reasonings is to make them as tangible as those of the mathematicians, so that we can find our error at a glance, and when there are disputes among persons we can simply say, "Let us calculate... to see who is right."

—LEIBNIZ, *The Art of Discovery*

15.0 Introduction

As an implementation of logic programming, PROLOG makes many interesting contributions to AI problem solving. These include its *declarative semantics*, a means of directly expressing problem relationships in AI, as well as with built-in unification, some high-powered techniques for pattern matching and search. We address many of the important issues of PROLOG and logic programming in this chapter.

In Section 15.1 we present the basic PROLOG syntax and several simple programs. These programs demonstrate the use of the predicate calculus as a representation language. We show how to monitor the PROLOG environment and demonstrate the use of the *cut* with PROLOG's built in depth-first search.

In Section 15.2 we create *abstract data types* (ADTs) in PROLOG. These ADTs include *stacks*, *queues*, and *priority queues*, which are then used to build a production system in Section 15.3 and to design control structures for the search algorithms of Chapters 3, 4, and 7 in Section 15.4. In Section 15.5 we create a *planner*, after the material presented in Section 8.4. In Section 15.6 we introduce *meta-predicates*, predicates whose domains of interpretation are PROLOG expressions themselves. For example, atom(X) succeeds if X is bound to an atom. Meta-predicates may be used for imposing type constraints on

PROLOG interpretations. In Section 15.7 meta-predicates are used for building *meta-interpreters* in PROLOG. Meta-interpreters are used to build a PROLOG interpreter in PROLOG, as well as to build interpreters for rule chaining and inheritance searches.

In Section 15.8 we demonstrate PROLOG as a language for machine learning, with examples of version space search and explanation-based learning from Chapter 10. In Section 15.9 we build a recursive descent semantic net parser, based on ideas developed in Chapter 14. The chapter ends with the discussion of the general issues of programming in logic and procedural versus declarative problem solving.

15.1 Syntax for Predicate Calculus Programming

15.1.1 Representing Facts and Rules

Although there are numerous dialects of PROLOG, the syntax used throughout this text is the original Warren and Pereira C-PROLOG (Clocksin and Mellish 2003). To simplify our presentation of PROLOG, our version of predicate calculus syntax in Chapter 2 used many PROLOG conventions. There are, however, a number of differences between PROLOG and predicate calculus syntax. In C-PROLOG, for example, the symbol :- replaces the ← of first-order predicate calculus. Other symbols differ from those used in Chapter 2:

ENGLISH	PREDICATE CALCULUS	PROLOG
and	$\wedge$	,
or	$\vee$	;
only if	$\leftarrow$	:-
not	$\neg$	not

As in Chapter 2, predicate names and bound variables are expressed as a sequence of alphanumeric characters beginning with an alphabetic. Variables are represented as a string of alphanumeric characters beginning (at least) with an uppercase alphabetic. Thus:

```
likes(X, susie).
```

or, better,

```
likes(Everyone, susie).
```

could represent the fact that "everyone likes Susie." Or,

```
likes(george, Y), likes(susie, Y).
```

could represent the set of things (or people) that are liked by both George and Susie.

Similarly, suppose it was desired to represent in PROLOG the following relationships: "George likes Kate and George likes Susie." This could be stated as:

```
likes(george, kate), likes(george, susie).
```

Likewise, "George likes Kate or George likes Susie":

```
likes(george, kate); likes(george, susie).
```

Finally, "George likes Susie if George does not like Kate":

```
likes(george, susie) :- not(likes(george, kate)).
```

These examples show how the predicate calculus connectives $\wedge$, $\vee$, $\neg$, and $\leftarrow$ are expressed in PROLOG. The predicate names (likes), the number or order of parameters, and even whether a given predicate always has the same number of parameters are determined by the design requirements (the implicit "semantics") of the problem. There are no expressive limitations, other than the syntax of well-formed formulae, in the language.

A PROLOG program is a set of specifications in the first-order predicate calculus describing the objects and relations in a problem domain. The set of specifications is referred to as the *database* for that problem. The PROLOG interpreter responds to questions about this set of specifications. Queries to the database are patterns in the same logical syntax as the database entries. The PROLOG interpreter uses pattern-directed search to find whether these queries logically follow from the contents of the database.

The interpreter processes queries, searching the database in left to right depth-first order to find out whether the query is a logical consequence of the database of specifications. PROLOG is primarily an interpreted language. Some versions of PROLOG run in interpretive mode only, while others allow compilation of part or all of the set of specifications for faster execution. PROLOG is an interactive language; the user enters queries in response to the PROLOG prompt: ?-.

Suppose that we wish to describe a "world" consisting of George's, Kate's, and Susie's likes and dislikes. The database might contain the following set of predicates:

```
likes(george, kate).
likes(george, susie).
likes(george, wine).
likes(susie, wine).
likes(kate, gin).
likes(kate, susie).
```

This set of specifications has the obvious interpretation, or mapping, into the world of George and his friends. This world is a model for the database (Section 2.3). The interpreter may then be asked questions:

```
?- likes(george, kate).
yes
```

```
?- likes(kate, susie).
yes
?- likes(george, X).
X = kate
;
X = susie
;
X = wine
;
no
?- likes(george, beer).
no
```

Note several things in these examples. First, in the request likes(george, X), successive user prompts (;) cause the interpreter to return all the terms in the database specification that may be substituted for the X in the query. They are returned in the order in which they are found in the database: kate before susie before wine. Although it goes against the philosophy of nonprocedural specifications, a determined order of evaluation is a property of most interpreters implemented on sequential machines. The PROLOG programmer must be aware of the order in which PROLOG searches entries in the database.

Also note that further responses to queries are produced when the user prompts with the ; (or). This forces a backtrack on the most recent result. Continued prompts force PROLOG to find all possible solutions to the query. When no further solutions exist, the interpreter responds no.

The above example also illustrates the *closed world assumption* or *negation as failure*. PROLOG assumes that "anything is false whose opposite is not provably true." In the query likes(george, beer), the interpreter looks for the predicate likes(george, beer) or some rule that could establish likes(george, beer). Failing this, the request is false. Thus, PROLOG assumes that all knowledge of the world is present in the database.

The closed world assumption introduces a number of practical and philosophical difficulties in the language. For example, failure to include a fact in the database often means that its truth is unknown; the closed world assumption treats it as false. If a predicate were omitted or there were a misspelling, such as likes(george, beeer), the response remains no. The negation-as-failure issue is a very important topic in AI research. Though negation as failure is a simple way to deal with the problem of unspecified knowledge, more sophisticated approaches, such as multivalued logics (true, false, unknown) and nonmonotonic reasoning (see Section 9.1), provide a richer interpretive context.

The PROLOG expressions used in the database above are examples of *fact* specifications. PROLOG also lets us define *rules* to describe relationships between facts using the logical implication, :- . In creating a PROLOG rule, only one predicate is permitted on the left-hand side of the if symbol, :-; this predicate must be a *positive literal*, which means it cannot be negated (Section 13.3). All predicate calculus expressions that contain implication or equivalence relationships ($\leftarrow$, $\rightarrow$, and $\leftrightarrow$) must be reduced to this form, referred to as *Horn clause logic*. In Horn clause form, the left-hand side (conclusion) of an implication must be a single positive literal. The *Horn clause calculus* is equivalent to the full first-order predicate calculus for proofs by refutation, see details in Chapter 13.

Suppose we add to the specifications of the previous database a rule for determining whether two people are friends. This may be defined:

```
friends(X, Y) :- likes(X, Z), likes(Y, Z).
```

This expression might be interpreted as "X and Y are friends if there exists a Z such that X likes Z and Y likes Z ." Two issues are important here. First, because neither the predicate calculus nor PROLOG has global variables, the scope (extent of definition) of X, Y, and Z is limited to the friends rule. Second, values bound to, or unified with, X, Y, and Z are consistent across the entire expression. The treatment of the friends rule by the PROLOG interpreter is seen in the following example.

With the friends rule added to the set of specifications of the preceding example, we can query the interpreter:

```
?- friends(george, susie).
yes
```

To solve the query, PROLOG searches the database using the backtrack algorithm presented in Chapters 3 and 6. The query friends(george, susie) is matched or unified with the conclusion of the rule friends(X, Y) :- likes(X, Z), likes(Y, Z), with X as george and Y as susie. The interpreter looks for a Z such that likes(george, Z) is true. This is first attempted using the first fact in the database, with Z as kate.

The interpreter then tries to determine whether likes(susie, kate) is true. When it is found to be false, using the closed world assumption, this value for Z (kate) is rejected. The interpreter then backtracks to find a second value for Z in likes(george, Z).

likes(george, Z) then matches the second clause in the database, with Z bound to susie. The interpreter then tries to match likes(susie, susie). When this also fails, the interpreter goes back to the database (backtracks) for yet another value for Z. This time wine is found in the third predicate, and the interpreter goes on to show that likes(susie, wine) is true. In this case wine is the binding that ties george and susie. PROLOG tries to match goals with patterns in the order in which the patterns are entered in the database.

It is important to state the relationship between universal and existential quantification in the predicate calculus and the treatment of variables in a PROLOG program. When a variable is placed in the specifications of a PROLOG database, the variable is assumed to be universally quantified. For example, likes(susie, Y) means, according to the semantics of the previous examples, "Susie likes everyone." In the course of interpreting some query, any term, or list or predicate, may be bound to Y. Similarly, in the rule friends(X, Y) :- likes(X, Z), likes(Y, Z), any X, Y, and Z that meet the specifications of the expression are acceptable variable bindings.

To represent an existentially quantified variable in PROLOG, we may take two approaches. First, if the existential value of a variable is known, that value may be entered directly into the database. Thus, likes(george, wine) is an instance of likes(george, Z) and may be thus entered into the database, as it was in the previous examples.

Second, to find an instance of a variable that makes an expression true, we query the interpreter. For example, to find whether a Z exists such that likes(george, Z) is true, we put this query directly to the interpreter. It will find whether a value of Z exists under which the expression is true. Some PROLOG interpreters find all existentially quantified values; C-PROLOG requires repeated user prompts (;) to get all values.

15.1.2 Creating, Changing, and Monitoring the PROLOG Environment

In creating a PROLOG program the database of specifications is created first. In an interactive environment the predicate assert adds new predicates to specifications. Thus:

```
?- assert(likes(david, sarah)).
```

adds this predicate to the computing specifications. Now, with the query:

```
?- likes(david, X).
X = sarah.
```

is returned. assert allows further control in adding new specifications to the database: asserta(P) asserts the predicate P at the beginning of all the predicates P, and assertz(P) adds P at the end of all the predicates named P. This is important for search priorities and building heuristics. To remove a predicate P from the database retract(P) is used. (It should be noted that in many PROLOGs assert can be unpredictable in that the exact entry time of the new predicate into the environment can vary depending on what other things are going on, affecting both the indexing of asserted clauses and backtracking.)

It soon becomes tedious to create a set of specifications using the predicates assert and retract. Instead, the programmer takes her favorite editor and creates a file containing all the PROLOG specifications. Once this file is created (let's call it myfile) and PROLOG is called, then the file is placed in the database by the PROLOG command consult. Thus:

```
?- consult(myfile).
yes
```

adds the predicates in myfile to the database. A short form of the consult predicate, and better for adding multiple files to the database, uses the list notation, to be seen shortly:

```
?- [myfile].
yes
```

The predicates read and write are important for user communication. read(X) takes the next term from the current input stream and binds it to X. Input expressions are terminated with a ".". write(X) puts X in the output stream. If X is unbound then an integer preceded by an underline is printed (_69). This integer represents the internal bookkeeping on variables necessary in a theorem-proving environment (see how variables are *standardized apart* in Section 13.2.2).

The PROLOG predicates see and tell are used to read information from and place information into files. see(X) opens the file X and defines the current input stream as originating in X. If X is not bound to an available file see(X) fails. Similarly, tell(X) opens a file for the output stream. If no file X exists, tell(X) creates a file named by the bound value of X. seen(X) and told(X) close the respective files.

A number of PROLOG predicates are important in helping us keep track of the state of the PROLOG database as well as the state of computing about the database; the most important of these are listing, trace, and spy. If we use listing(predicate_name) where predicate_name is the name of a predicate, such as member (Section 15.1.3), all the clauses with that predicate name in the database are returned by the interpreter. Note that the number of arguments of the predicate is not indicated; in fact, all uses of the predicate, regardless of the number of arguments, are returned.

trace allows the user to monitor the progress of the PROLOG interpreter. This monitoring is accomplished by printing to the output file every goal that PROLOG attempts, which is often more information than the user wants to have. The tracing facilities in many PROLOG environments are rather cryptic and take some study and experience to understand. The information available in a trace of a running PROLOG program usually includes the following:

1. The depth level of recursive calls (marked left to right on line).
2. When a goal is tried for the first time (sometimes call is used).
3. When a goal is successfully satisfied (with an exit).
4. When a goal has further matches possible (a retry).
5. When a goal fails because all attempts to satisfy it have failed (fail is often used).
6. The goal notrace stops the exhaustive tracing.

When a more selective trace is required the goal spy is useful. This predicate usually takes a predicate name as argument but sometimes is defined as a prefix operator where the predicate to be monitored is listed after the operator. Thus, spy member causes the interpreter to print to output all uses of the predicate member. spy can also take a list of predicates followed by their arities: spy[member/2, append/3] sets monitoring of the interpreter on all uses of the goals member with two arguments and append with three. nospy removes these spy points.

15.1.3 Lists and Recursion in PROLOG

The previous subsections presented PROLOG syntax in several simple examples. These examples introduced PROLOG as an engine for computing with predicate calculus expressions (in Horn clause form). This is consistent with all the principles of predicate calculus inference presented in Chapter 2. PROLOG uses unification for pattern matching and returns the bindings that make an expression true. These values are unified with the variables in a particular expression and are not bound in the global environment.

Recursion is the primary control mechanism for PROLOG programming. We will demonstrate this with several examples. But first we consider some simple list-processing examples. The list is a data structure consisting of ordered sets of elements (or, indeed, lists). Recursion is the natural way to process the list structure. Unification and recursion come together in list processing in PROLOG. The list elements themselves are enclosed by brackets [ ] and are separated by commas. Examples of PROLOG lists are:

```
[1, 2, 3, 4]
[[george, kate], [allen, amy], [don, pat]]
[tom, dick, harry, fred]
[ ]
```

The first elements of a list may be separated from the tail of the list by the bar operator, |. The tail of a list is the list with its first element removed. For instance, when the list is [tom,dick,harry,fred], the first element is tom and the tail is the list [dick, harry, fred]. Using the vertical bar operator and unification, we can break a list into its components:

If [tom, dick, harry, fred] is matched to [X | Y], then X = tom and Y = [dick, harry, fred].

If [tom,dick,harry,fred] is matched to pattern [X, Y | Z], then X = tom , Y = dick , and Z = [harry, fred].

If [tom, dick, harry, fred] is matched to [X, Y, Z | W], then X = tom, Y = dick, Z = harry, and W = [fred].

If [tom, dick, harry, fred] is matched to [W, X, Y, Z | V], then W = tom, X = dick, Y = harry, Z = fred, and V = [ ] .

[tom, dick, harry, fred] will not match [V, W, X, Y, Z | U] .

[tom, dick, harry, fred] will match [tom, X | [harry, fred]], to give X = dick.

Besides "tearing lists apart" to get at particular elements, unification can be used to "build" the list structure. For example, if X = tom, Y = [dick], and L unifies with [X | Y], then L will be bound to [tom, dick]. Thus terms separated by commas before the | are all elements of the list, and the structure after the | is always a list, the tail of the list.

Let's take a simple example of recursive processing of lists: the member check. We define a predicate to determine whether an item, represented by X, is in a list. This predicate member takes two arguments, an element and a list, and is true if the element is a member of the list. For example:

```
?- member(a, [a, b, c, d, e]).
yes
?- member(a, [1, 2, 3, 4]).
no
?- member(X, [a, b, c]).
X = a
;
X = b
;
X = c
;
no
```

To define member recursively, we first test if X is the first item in the list:

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

This tests whether X and the first element of the list are identical. If they are not, then it is natural to check whether X is an element of the rest (T) of the list. This is defined by:

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

The two lines of PROLOG for checking list membership are then:

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

This example illustrates the importance of PROLOG's built-in order of search with the terminating condition placed before the recursive call, to be tested before the algorithm recurs. If the order of the predicates is reversed, the terminating condition may never be checked. We now trace member(c,[a,b,c]), with numbering:

```
1: member(X, [X | T]).
2: member(X, [Y | T]) :- member(X, T).

?- member(c, [a, b, c]).
    call 1. fail, since c ≠ a
    call 2. X = c, Y = a, T = [b, c], member(c, [b,c])?
        call 1. fail, since c ≠ b
        call 2. X = c, Y = b, T = [c], member(c, [c])?
            call 1. success, c = c
        yes (to second call 2.)
    yes (to first call 2.)
yes
```

Good PROLOG style suggests the use of *anonymous variables*. These serve as an indication to the programmer and interpreter that certain variables are used solely for pattern-matching purposes, with the variable binding itself not part of the computation process. Thus, when we test whether the element X is the same as the first item in the list we usually say: member(X, [X|_]). The use of the _ indicates that even though the tail of the list plays a crucial part in the unification of a query, the content of the tail of the list is unimportant. In the member check the anonymous variable should be used in the recursive statement as well, where the value of the head of the list is unimportant:

```
member(X, [X | _]).
member(X, [_ | T]) :- member(X, T).
```

Writing out a list one element to a line is a nice exercise for understanding both lists and recursive control. Suppose we wish to write out the list [a,b,c,d]. We could define the recursive command:

```
writelist([ ]).
writelist([H | T]) :- write(H), nl, writelist(T).
```

This predicate writes one element of the list on each line, as nl requires the output stream controller to begin a new line. If we wish to write out a list in reversed order the recursive predicate must come before the write command. This guarantees that the list is traversed to the end before any element is written. At that time the last element of the list is written followed by each preceding element as the recursive control comes back up to the top. A reverse write of a list would be:

```
reverse_writelist([ ]).
reverse_writelist([H | T]) :- reverse_writelist(T), write(H), nl.
```

The reader should run writelist and reverse_writelist with trace to observe the behavior of these predicates.

15.1.4 Recursive Search in PROLOG

In Section 6.2 we introduced the 3×3 knight's tour problem for the predicate calculus. We represented the board squares for the knight moves like this:

1	2	3
4	5	6
7	8	9

The legal moves are represented in PROLOG using a move predicate. The path predicate defines an algorithm for finding a path of zero or more moves between its arguments. Note that path is defined recursively:

```
move(1, 6).      move(3, 4).      move(6, 7).      move(8, 3).
move(1, 8).      move(3, 8).      move(6, 1).      move(8, 1).
move(2, 7).      move(4, 3).      move(7, 6).      move(9, 4).
move(2, 9).      move(4, 9).      move(7, 2).      move(9, 2).

path(Z, Z).
path(X, Y) :- move(X, W), not(been(W)), assert(been(W)), path(W, Y).
```

This definition of path is a PROLOG implementation of the algorithm defined in Chapter 6. As noted above, assert is a built-in PROLOG predicate that always succeeds and has the side effect of placing its argument in the database of specifications. The been predicate is used to record previously visited states and avoid loops.

This use of the been predicate violates the program designer's goal of creating predicate calculus specifications that do not use global variables. Thus been(3), when asserted into the database, is indeed a fact available to any other procedure in the database and, as such, has global extension. Even more important, creating global structures to alter program control violates the basic tenet of the production system model, where the logic (of problem specifications) is kept separate from the control of the program. Here been structures were created as global specifications to modify the execution of the program itself.

As we proposed in Chapter 3, a list may be used to keep track of visited states and thus keep the path call from looping. The member predicate is used to detect duplicate states (loops). This approach remedies the problems of using global been(W) assertions. The PROLOG-based specification of the following clauses exactly implements the depth-first graph search with the backtracking algorithm of Chapters 3 and 6:

```
path(Z, Z, L).
path(X, Y, L) :- move(X, Z), not(member(Z, L)), path(Z, Y, [Z | L]).
```

using the member predicate as defined previously.

The third parameter of path is the local variable representing the list of states that have already been visited. When a new state is generated (using the move predicate) and this state is not already on the list of visited states, not(member(Z, L)), it is placed on the front of the state list [Z | L] for the next path call.

It should be noted that all the parameters of path are local and their current values depend on where they are called in the graph search. Each recursive call adds a state to this list. If all continuations from a certain state fail, then that particular path call fails. When the interpreter backs up to the parent call, the third parameter, representing the list of states visited, has its previous value. Thus, states are added to and deleted from this list as the backtracking search moves through the graph.

When the path call finally succeeds, the first two parameters are identical. The third parameter is the list of states visited on the solution path, in reverse order. Thus we can print out the steps of the solution. The PROLOG specification for the knight's tour problem using lists and a depth-first search employing backtrack may be obtained by using this definition of path with the move specifications and member predicates just presented.

The call to the PROLOG interpreter path(X,Y,[X]), where X and Y are replaced by numbers between 1 and 9, finds a path from state X to state Y, if the path exists. The third parameter initializes the path list with the starting state X. Note that there is no typing distinction in PROLOG: the first two parameters are any representation of states in the problem space and the third is a list of states. Unification makes this generalization of pattern matching across data types possible. Thus, path is a general depth-first search algorithm that may be used with any graph. In Section 15.3 we use this to implement a production system solution to the farmer, wolf, goat, and cabbage problem, with state specifications replacing square numbers in the call to path.

We now present the solution for the 3×3 knight's tour. It is left as an exercise to solve the full 8×8 knight's tour problem in PROLOG. (See exercises in Chapters 15 and 16.) For this trace we refer to the two parts of the path algorithm by number:

```
1. is path(Z, Z, L).
2. is path(X, Y, L) :- move(X, Z), not(member(Z, L)), path(Z, Y, [Z | L]).
?- path(1, 3, [1]).
```

```
path(1, 3, [1]) attempts to match 1.  fail 1 ≠ 3.
path(1, 3, [1]) matches 2. X is 1, Y is 3, L is [1]
  move(1, Z) matches Z as 6, not(member(6, [1])) is true, call path(6, 3, [6,1])
    path(6, 3, [6, 1]) attempts to match 1.  fail 6 ≠ 3.
    path(6, 3, [6, 1]) matches 2. X is 6, Y is 3, L is [6, 1].
      move(6, Z) matches Z as 7, not(member(7, [6, 1])) is true, path(7, 3, [7, 6, 1])
      path(7, 3, [7, 6, 1]) attempts to match 1. fail 7 ≠ 3.
      path(7, 3, [7, 6, 1]) matches 2. X is 7, Y is 3, L is [7, 6, 1].
        move(7, Z) is Z = 6, not(member(6, [7, 6, 1])) fails, backtrack!
        move(7, Z) is Z = 2, not(member(2, [7, 6, 1])) true, path(2, 3, [2, 7, 6, 1])
        path call attempts 1, fail, 2 ≠ 3.
        path matches 2, X is 2, Y is 3, L is [2, 7, 6, 1]
          move matches Z as 7, not(member(...)) fails, backtrack!
          move matches Z as 9, not(member(...)) true, path(9, 3, [9, 2, 7, 6, 1])
          path fails 1, 9 ≠ 3.
          path matches 2, X is 9, Y is 3, L is [9, 2, 7, 6, 1]
            move is Z = 4, not(member(...)) true, path(4, 3, [4, 9, 2, 7, 6, 1])
            path fails 1, 4 ≠ 3.
            path matches 2, X is 4, Y is 3, L is [4, 9, 2, 7, 6, 1]
              move Z = 3, not(member(...)) true, path(3, 3, [3, 4, 9, 2, 7, 6, 1])
                path attempts 1, true, 3 = 3, yes
          yes
        yes
      yes
    yes
  yes
yes
```

In summary, the recursive path call is a *shell* or general control structure for search in a graph: in path(X, Y, L), X is the present state; Y is the goal state. When X and Y are identical, the recursion terminates. L is the list of states on the current path to state Y, and as each new state Z is found with the call move(X, Z) it is placed on the list: [Z | L]. The state list is checked, using not(member(Z, L)), to be sure the path does not loop.

The difference between the state list L in the path call above and the closed set in Chapter 3 is that closed records all states visited, while the state list L keeps track of only the present path. It is straightforward to expand the record keeping in the path call to record all visited states and we do this in Section 15.4.

15.1.5 The Use of Cut to Control Search in PROLOG

The *cut* is represented by an exclamation point, !. The syntax for cut is that of a goal with no arguments, that has several side effects: first, when originally encountered it always succeeds, and second, if it is "failed back to" in backtracking, it causes the entire goal in which it is contained to fail. For a simple example of the effect of the cut, recall the two-move path call from the knight's tour example. The predicate path2 could be created:

```
path2(X, Y) :- move(X, Z), move(Z, Y).
```

(There is a two-move path between X and Y if there exists an intermediate stop Z between them.) For this example, assume part of the knight's database:

```
move(1, 6).
move(1, 8).
move(6, 7).
move(6, 1).
move(8, 3).
move(8, 1).
```

The interpreter is asked to find all the two-move paths from 1; there are four answers:

```
?- path2(1, W).
W = 7
;
W = 1
;
W = 3
;
W = 1
;
no
```

When path2 is altered with cut, only two answers result:

```
path2(X, Y) :- move(X, Z), !, move(Z, Y).

?- path2(1, W).
W = 7
;
W = 1
;
no
```

This happens because variable Z takes on only one value (the first value it is bound to), namely 6. Once the first subgoal succeeds, Z is bound to 6 and the cut is encountered. This prohibits further backtracking to the first subgoal and any further bindings for Z.

There are several uses for the cut in programming. First, as this example demonstrated, it allows the programmer to control explicitly the shape of the search tree. When further (exhaustive) search is not required, the tree can be explicitly pruned at that point. This allows PROLOG code to have the flavor of function calling: when one set of values (bindings) is "returned" by a PROLOG predicate (or set of predicates) and the cut is encountered, the interpreter does not search for other unifications. If that set of values does not lead on to a solution then no further values are attempted.

A second use of cut controls recursion. For example in the path call:

```
path(Z, Z, L).
path(X, Z, L) :- move(X, Y), not(member(Y, L)), path(Y, Z, [Y|L]), !.
```

the addition of cut means that (at most) one solution to the graph search is produced. Only one solution is produced because further solutions occur after the clause path(Z, Z, L) is satisfied. If the user asks for more solutions, path(Z, Z, L) fails, and the second path call is reinvoked to continue the (exhaustive) search of the graph. When the cut is placed after the recursive path call, the call cannot be reentered (backed into) for further search.

Important side effects of the cut are to make the program run faster and to conserve memory locations. When cut is used within a predicate, the pointers in memory needed for backtracking to predicates to the left of the cut are not created. This is, of course, because they will never be needed. Thus, cut produces the desired solution, and only the desired solution, with more efficient use of memory.

The cut can also be used with recursion to reinitialize the path call for further search within the graph. This will be demonstrated with the general search algorithms presented in Section 15.3. For this purpose we also need to develop several abstract data types.

15.2 Abstract Data Types (ADTs) in PROLOG

Programming in any environment is enhanced by procedural abstractions and information hiding. Because the set, stack, queue, and priority queue data structures were the support constructs for the graph search algorithms of Chapters 3, 4, and 6, we build them in PROLOG in the present section and then use them in the design of the PROLOG search algorithms presented later in this chapter.

Recursion, lists, and pattern matching, as emphasized throughout this book, are the primary tools for building and searching graph structures. These are the pieces with which we build our ADTs. All list handling and recursive processing that define the ADT are "hidden" within the ADT abstraction, quite different than the normal static data structure.

15.2.1 The ADT Stack

A *stack* is a linear structure with access at one end only. Thus all elements must be added to, pushed, and removed, popped, from the structure at that end of access. The stack is sometimes referred to as a last-in-first-out (LIFO) data structure. We saw its use with depth-first search in Section 3.2.3. The operators that we will define for a stack are:

1. Test whether the stack is empty.
2. Push an element onto the stack.
3. Pop, or remove, the top element from the stack.
4. Peek (often called Top) to see the top element on the stack without popping it.

5. Member_stack, which checks whether an element is in the stack.
6. Add_list, which adds a list of elements to the stack.

Operators 5 and 6 may be built from 1–4.

We now build these operators in PROLOG. As just noted, we use the list primitives:

1. empty_stack([ ]). This predicate can be used either to test a stack to see whether it is empty or to generate a new empty stack.

2–4. stack(Top, Stack, [Top | Stack]). This predicate performs the push, pop, and peek predicates depending on the variable bindings of its arguments. For instance, push produces a new stack as the third argument when the first two arguments are bound. Likewise, pop produces the top element of the stack when the third argument is bound to the stack. The second argument will then be bound to the new stack, once the top element is popped. Finally, if we keep the stack as the third argument, the first argument lets us peek at its top element.

5. member_stack(Element, Stack) :- member(Element, Stack). This allows us to determine whether an element is a member of the stack. Of course, the same result could be produced by creating a recursive call that peeked at the next element of the stack and then, if this element did not match Element, popped the stack. This would continue until the empty stack predicate was true.

6. add_list_to_stack(List, Stack, Result) :- append(List, Stack, Result). List is added to Stack to produce Result, a new stack. Of course, the same result could be obtained by popping List and pushing each element onto a temporary stack until empty stack is true of List. We then pop the temporary stack and push each element onto the Stack until empty stack is true of the temporary stack. append is described in detail in Section 15.10.

A final predicate for printing a stack in reverse order is reverse_print_stack. This is very useful when a stack has, in reversed order, the current path from the start state to the present state of the graph search. We will see several examples of this in the following subsections.

```
reverse_print_stack(S) :- empty_stack(S).
reverse_print_stack(S) :-
    stack(E, Rest, S),
    reverse_print_stack(Rest),
    write(E), nl.
```

15.2.2 The ADT Queue

A *queue* is a first-in-first-out (FIFO) data structure. It is often characterized as a list where elements are taken off (dequeued) from one end and added to (enqueued) at the other end. The queue was used for defining breadth-first search in Chapters 3 and 4. The queue operators are:

1. empty_queue([ ]). This predicate either tests whether a queue is empty or initializes a new empty queue.
2. enqueue(E, [ ], [E]).
   enqueue(E, [H | T], [H | Tnew]) :- enqueue(E, T, Tnew). This recursive predicate adds the element E to a queue, the second argument. The new augmented queue is the third argument.
3. dequeue(E, [E | T], T). This predicate produces a new queue, the third argument, that is the result of taking the next element, the first argument, off the original queue, the second argument.
4. dequeue(E, [E | T], _). This predicate lets us peek at the next element, E, of the queue.
5. member_queue(Element, Queue) :- member(Element, Queue). This tests whether Element is a member of Queue.
6. add_list_to_queue(List, Queue, Newqueue) :- append(Queue, List, Newqueue). This enqueues an entire list of elements.

Of course, 5 and 6 can be created using 1–4; append is presented in Section 15.10.

15.2.3 The ADT Priority Queue

A *priority queue* orders the elements of a regular queue so that each new entrant to the priority queue is placed in its sorted order. The dequeue operator removes the "best" sorted element from the priority queue. We used the priority queue in the design of best-first search in Chapter 4.

Because the priority queue is a sorted queue, many of its operators are the same as the queue operators, in particular, empty_queue, member_queue, dequeue (the "best" of the sorted elements will be next for the dequeue), and peek. enqueue in a priority queue is the insert_pq operator, as each new item is placed in its proper sorted order.

```
insert_pq(State, [ ], [State]) :- !.
insert_pq(State, [H | Tail], [State, H | Tail]) :-
    precedes(State, H).
insert_pq(State, [H | T], [H | Tnew]) :-
    insert_pq(State, T, Tnew).
precedes(X, Y) :- X < Y.                %order operator depends on types compared
```

The first argument of this predicate is the new element that is to be inserted. The second argument is the previous priority queue, and the third argument is the augmented priority queue. The precedes predicate checks that the order of elements is preserved.

Another priority queue operator is insert_list_pq. This predicate is used to merge an unsorted list or set of elements into the priority queue, as is necessary when adding the children of a state to the priority queue for best-first search (Chapter 4 and Section 15.4.3). insert_list_pq uses insert_pq to put each individual new item into the priority queue:

```
insert_list_pq([ ], L, L).
insert_list_pq([State | Tail], L, New_L) :-
    insert_pq(State, L, L2),
    insert_list_pq(Tail, L2, New_L).
```

15.2.4 The ADT Set

Finally, we describe the ADT set. A *set* is a collection of elements with no element repeated. Sets can be used for collecting all the children of a state or for maintaining closed in a search algorithm, as in Chapters 3 and 4. A set of elements, e.g., {a,b}, is represented as a list, [a,b], with order not important. The set operators include empty_set, member_set, delete_if_in, and add_if_not_in. We have operators for combining and comparing sets, including union, intersection, set_difference, subset, and equal_set.

```
empty_set([ ]).
member_set(E, S) :-
    member(E, S).
delete_if_in_set(E, [ ], [ ]).
delete_if_in_set(E, [E | T], T) :- !.
delete_if_in_set(E, [H | T], [H | T_new]) :-
    delete_if_in_set(E, T, T_new), !.
add_if_not_in_set(X, S, S) :-
    member(X, S), !.
add_if_not_in_set(X, S, [X | S]).
union([ ], S, S).
union([H | T], S, S_new) :-
    union(T, S, S2),
    add_if_not_in_set(H, S2, S_new),!.
subset([ ], _).
subset([H | T], S) :-
    member_set(H, S),
    subset(T, S).
intersection([ ], _, [ ]).
intersection([H | T], S, [H | S_new]) :-
    member_set(H, S),
    intersection(T, S, S_new), !.
intersection([_ | T], S, S_new) :-
    intersection(T, S, S_new), !.
set_difference([ ], _, [ ]).
set_difference([H | T], S, T_new) :-
    member_set(H, S),
    set_difference(T, S, T_new), !.
set_difference([H | T], S, [H | T_new]) :-
    set_difference(T, S, T_new), !.
equal_set(S1, S2) :-
    subset(S1, S2),
    subset(S2, S1).
```

15.3 A Production System Example in PROLOG

In this section we write a production system solution to the farmer, wolf, goat, and cabbage problem. The problem is stated as follows:

> A farmer with his wolf, goat, and cabbage come to the edge of a river they wish to cross. There is a boat at the river's edge, but, of course, only the farmer can row. The boat also can carry only two things (including the rower) at a time. If the wolf is ever left alone with the goat, the wolf will eat the goat; similarly, if the goat is left alone with the cabbage, the goat will eat the cabbage. Devise a sequence of crossings of the river so that all four characters arrive safely on the other side of the river.

In the next paragraphs we present a production system solution to this problem. First, we observe that the problem may be represented as a search through a graph. To do this we consider the possible moves that might be available at any time in the solution process. Some of these moves are eventually ruled out because they produce states that are unsafe (something will be eaten).

For the moment, suppose that all states are safe, and simply consider the graph of possible states. The boat can be used in four ways: to carry the farmer and wolf, the farmer and goat, the farmer and cabbage, and the farmer alone. A state of the world is some combination of the characters on the two banks. Several states of the search are represented in Figure 15.1. States of the world may be represented using the predicate, state(F, W, G, C), with the location of the farmer as first parameter, location of the wolf as second parameter, the goat as third, and the cabbage as fourth. We assume that the river runs "north to south" and that the characters are on either the east, e, or west, w, bank. Thus, state(w, w, w, w) has all characters on the west bank to start the problem.

It must be pointed out that these choices are conventions that have been arbitrarily chosen by the authors. Indeed, as researchers in AI continually point out, the selection of an appropriate representation is often the most critical aspect of problem solving. These conventions are selected to fit the predicate calculus representation in PROLOG. Different states of the world are created by different crossings of the river, represented by changes in the values of the parameters of the state predicate as in Figure 15.1. Other representations are certainly possible.

We now describe a general graph for this river-crossing problem. For the time being, we ignore the fact that some states are unsafe. In Figure 15.2 we see the beginning of the graph of possible moves back and forth across the river. Since the farmer always rows, it is not necessary to have a separate representation for the location of the boat. Figure 15.2 represents part of the graph that is to be searched for a solution path.

The recursive path call previously described provides the control mechanism for the production system search. The production rules are the rules for changing state in the search. We define these as move rules in PROLOG form.

Because PROLOG uses Horn clauses, a production system designed in PROLOG must either represent production rules directly in Horn clause form or translate rules to this format. We take the former option here (and show how *if... then...* rules are changed to Horn clauses in Section 13.2). Horn clauses require that the pattern for the present state

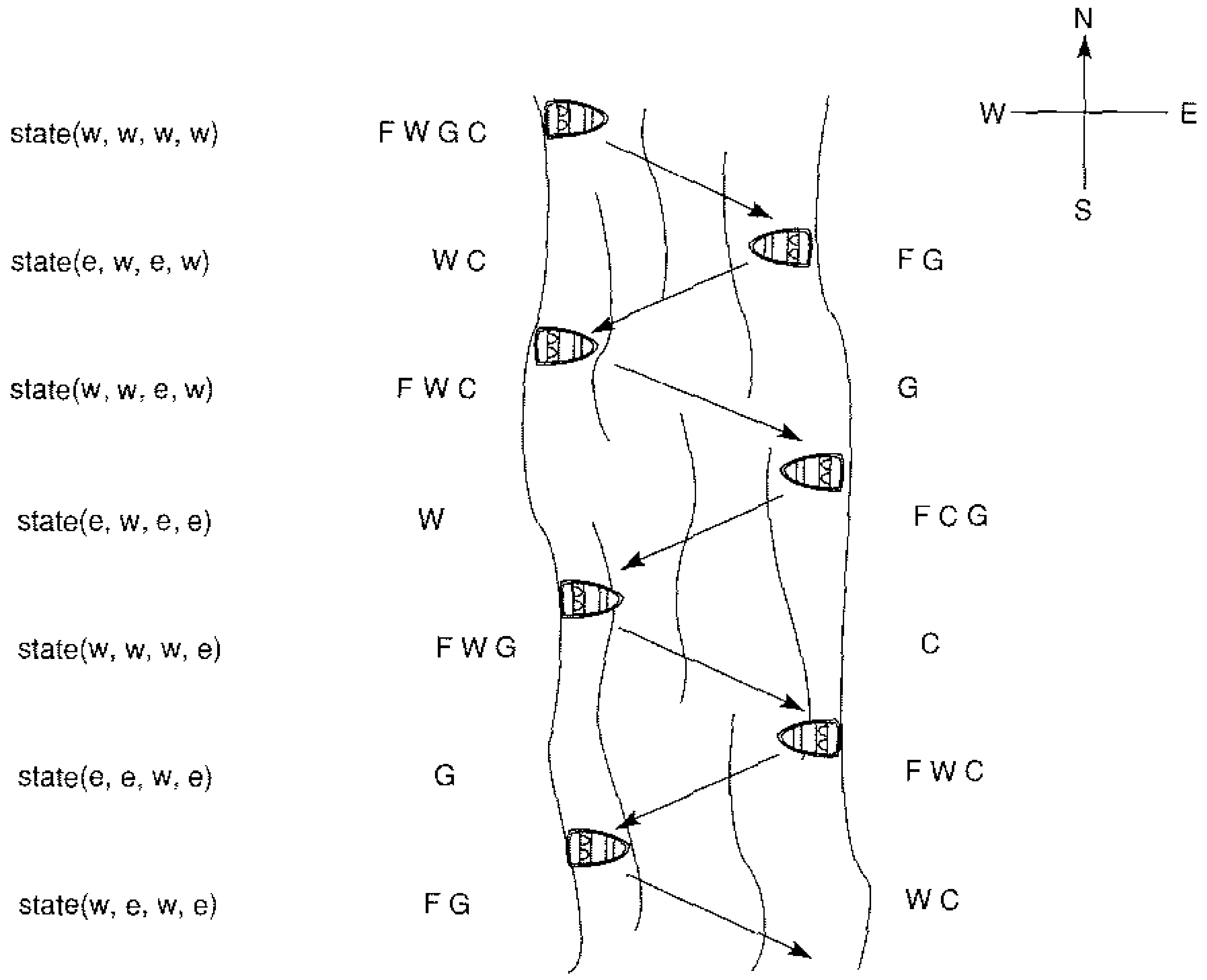

Figure 15.1 Sample crossings for the farmer, wolf, goat, and cabbage problem.

and the pattern for the next state both be placed in the head of the Horn clause, or to the left of :-. These are the arguments to the move predicate. The conditions that the production rule requires to fire and return the next state are placed to the right of :-. As shown in the following example, these conditions can also be expressed as unification constraints.

The first rule we define is for the farmer to take the wolf across the river. This rule must account for both the transfer from east to west and the transfer from west to east, and it must not be applicable when the farmer and wolf are on opposite sides of the river. Thus, it must transform state(e, e, G, C) to state(w, w, G, C) and state(w, w, G, C) to state(e, e, G, C). It must also fail for state(e, w, G, C) and state(w, e, G, C). The variables G and C represent the fact that the third and fourth parameters can be bound to either e or w. Whatever their values, they remain the same after the move of the farmer and wolf. Some of the states produced may indeed be "unsafe."

The following move rule operates only when the farmer and wolf are in the same location and takes them to the opposite side of the river. Note that the goat and cabbage do not change their present location (whatever it might be).

```
move(state(X, X, G, C), state(Y, Y, G, C)) :- opp(X, Y).
opp(e, w).
opp(w, e).
```

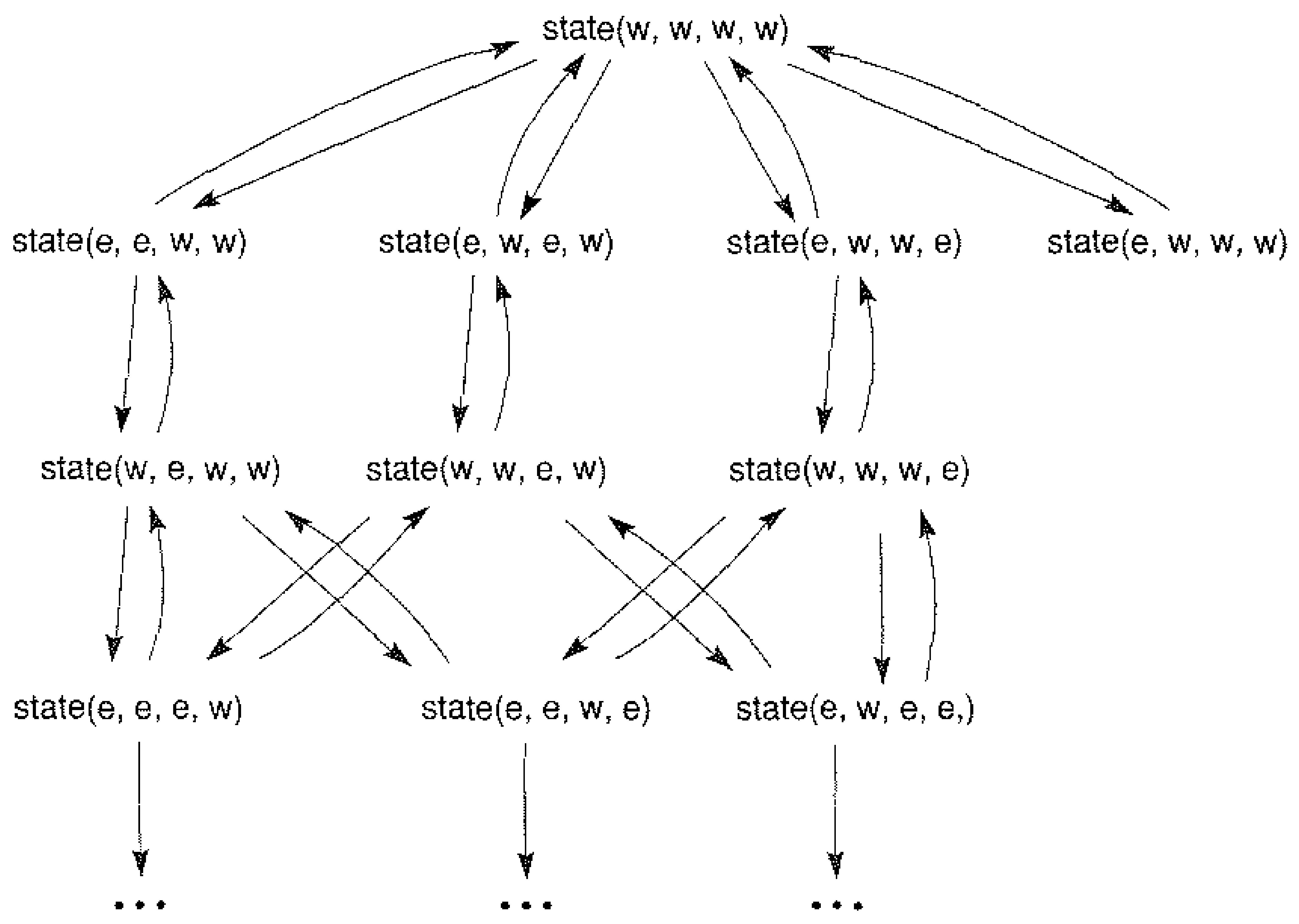

Figure 15.2 Portion of the state space graph of the farmer, wolf, goat, and cabbage problem, including unsafe states.

This rule fires when a state (the present location in the graph) is presented to the first parameter of move in which the farmer and wolf are at the same location. When the rule fires, a new state, the second parameter of move, is produced with the value of X opposite, opp, the value of Y. Two conditions are satisfied to produce the new state: first, that the values of the first two parameters are the same and, second, that both of their new locations are opposite their old.

The first condition was checked implicitly in the unification process, in that move is not even called unless the first two parameters are the same. This test may be done explicitly by using the following rule:

```
move(state(F, W, G, C), state(Z, Z, G, C)) :- F = W, opp(F, Z).
```

This equivalent move rule first tests whether F and W are the same and, only if they are (on the same side of the river), assigns the opposite value of F to Z. Note that PROLOG can do "assignment" by the binding of variable values in unification. Bindings are shared by all occurrences of a variable in a clause, and the scope of a variable is limited to the clause in which it occurs.

Pattern matching, a powerful tool in AI programming, is especially important in pruning search. States that do not fit the patterns in the rule are automatically pruned. In this sense, the first version of the move rule offers a more efficient representation because unification does not even consider the state predicate unless its first two parameters are identical.

Next, we create a predicate to test whether each new state is safe, so that nothing is eaten in the process of crossing the river. Again, unification plays an important role in this definition. Any state where the second and third parameters are the same and opposite the first parameter is unsafe; the wolf eats the goat. Alternatively, if the third and fourth parameters are the same and opposite the first parameter, the state is unsafe: the goat eats the cabbage. These unsafe situations may be represented with the following rules.

```
unsafe(state(X, Y, Y, C)) :- opp(X, Y).
unsafe(state(X, W, Y, Y)) :- opp(X, Y).
```

Several points should be mentioned here. First, if a state is to be not unsafe (i.e., safe), according to the definition of not in PROLOG, neither of these unsafe predicates can be true. Thus, neither of these predicates can unify with the current state or, if they do unify, their conditions must not be satisfied. Second, not in PROLOG is not exactly equivalent to the logical ¬ of the first-order predicate calculus; not is rather "negation by failure of its opposite." The reader should test a number of states to verify that unsafe does what it is intended to do. Now, not unsafe is added to the previous production rule:

```
move(state(X, X, G, C), state(Y, Y, G, C)) :-
     opp(X, Y), not(unsafe(state(Y, Y, G, C))).
```

The not unsafe test calls unsafe, as mentioned above, to see whether the generated state is an acceptable new state in the search. When all criteria are met, including the check in the path algorithm that the new state is not a member of the visited-state list, path is (recursively) called on this state to go deeper into the graph. When path is called, the new state is added to the visited-state list.

In a similar fashion, we can create the three other production rules to represent the farmer taking the goat, cabbage, and himself across the river. We have added a writelist command to each production rule to print a trace of the current rule.

The reverse_print_stack command is used in the terminating condition of path to print out the final solution path. Finally, we add a fifth "pseudorule" that always fires, because no conditions are placed on it, when all previous rules have failed; it indicates that the path call is backtracking from the current state, and then it itself fails. This pseudorule is added to assist the user in seeing what is going on as the production system is running.

We now present the full production system program in PROLOG to solve the farmer, wolf, goat, and cabbage problem. The PROLOG predicates unsafe, writelist, and the ADT stack predicates of Section 15.2.1, must also be included:

```
move(state(X, X, G, C), state(Y, Y, G, C)) :-
     opp(X, Y), not(unsafe(state(Y, Y, G, C))),
     writelist(['try farmer takes wolf', Y, Y, G, C]).
move(state(X, W, X, C), state(Y, W, Y, C)) :-
     opp(X, Y), not(unsafe(state(Y, W, Y, C))),
     writelist(['try farmer takes goat', Y, W, Y, C]).
move(state(X, W, G, X), state(Y, W, G, Y)) :-
     opp(X, Y), not(unsafe(state(Y, W, G, Y))),
     writelist(['try farmer takes cabbage', Y, W, G, Y]).
```

```
move(state(X, W, G, C), state(Y, W, G, C)) :-
     opp(X, Y), not(unsafe(state(Y, W, G, C))),
     writelist(['try farmer takes self', Y, W, G, C]).
move(state(F, W, G, C), state(F, W, G, C)) :-
     writelist(['    BACKTRACK from:', F, W, G, C]), fail.
path(Goal, Goal, Been_stack) :-
     write('Solution Path Is: ' ), nl,
     reverse_print_stack(Been_stack).
path(State, Goal, Been_stack) :-
     move(State, Next_state),
     not(member_stack(Next_state, Been_stack)),
     stack(Next_state, Been_stack, New_been_stack),
          path(Next_state, Goal, New_been_stack), !.
opp(e, w).
opp(w, e).
```

The code is called by requesting go, which initializes the recursive path call. To make running the program easier, we can create a predicate, called test, that simplifies the input:

```
go(Start, Goal) :-
   empty_stack(Empty_been_stack),
   stack(Start, Empty_been_stack, Been_stack),
   path(Start, Goal, Been_stack).

test :- go(state(w,w,w,w), state(e,e,e,e)).
```

The algorithm backtracks from states that allow no further progress. You may also use trace to monitor the various variable bindings local to each call of path. It may also be noted that this program is a general program for moving the four creatures from any (legal) position on the banks to any other (legal) position, including asking for a path from the goal back to the start state. Other interesting features of production systems, including the fact that different orderings of the rules can produce different searches through the graph, are presented in the exercises. A partial trace of the execution of the program, showing only rules actually used to generate new states, is presented next:

```
?- test.
try farmer takes goat e w e w
try farmer takes self w w e w
try farmer takes wolf e e e w
try farmer takes goat w e w w
try farmer takes cabbage e e w e
try farmer takes wolf w w w e
try farmer takes goat e w e e
   BACKTRACK from e,w,e,e
   BACKTRACK from w,w,w,e
try farmer takes self w e w e
try farmer takes goat e e e e
```

```
Solution Path Is:
state(w,w,w,w)
state(e,w,e,w)
state(w,w,e,w)
state(e,e,e,w)
state(w,e,w,w)
state(e,e,w,e)
state(w,e,w,e)
state(e,e,e,e)
```

In summary, this PROLOG program implements a production system solution to the farmer, wolf, goat, and cabbage problem. The move rules make up the content of the production memory. The working memory is represented by the arguments of the path call. The production system control mechanism is defined by the recursive path call. Finally, the ordering of rules for generation of children from each state (conflict resolution) is determined by the order in which the rules are placed in the production memory.

15.4 Designing Alternative Search Strategies

As the previous subsection demonstrated, and as is made more precise in Section 15.7, PROLOG itself uses depth-first search with backtracking. We now show how the alternative search strategies of Chapters 3, 4, and 6 can be implemented in PROLOG. Our implementations of depth-first, breadth-first, and best-first search use open and closed lists to record states in the search. When search fails at any point we do not go back to the preceding values of open and closed. Instead, open and closed are updated within the path call and the search continues with these new values. The cut is used to keep PROLOG from storing the old versions of open and closed.

15.4.1 Depth-First Search Using the Closed List

Because the values of variables are restored when recursion backtracks, the list of visited states in the depth-first path algorithm of Section 15.3 records states only if they are on the current path to the goal. Although the testing each "new" state for membership in this list prevents loops, it still allows branches of the space to be reexamined if they are reached along paths generated earlier but abandoned at that time as unfruitful. A more efficient implementation keeps track of all the states that have ever been encountered. This more complete collection of states made up the list called closed in Chapter 3, and Closed_set in the following algorithm.

Closed_set holds all states on the current path plus the states that were rejected when the algorithm backtracked out of them; thus, it no longer represents the path from the start to the current state. To capture this path information, we create the ordered pair [State, Parent] to keep track of each state and its parent; the Start state is represented by [Start, nil]. These state–parent pairs will be used to re-create the solution path from the Closed_set.

We now present a shell structure for depth-first search in PROLOG, keeping track of both open and closed and checking each new state to be sure it was not previously visited. path has three arguments, the Open_stack, Closed_set, maintained as a set, and the Goal state. The current state, State, is the next state on the Open_stack. The stack and set operators are found in Section 15.2.

Search starts by a go predicate that initializes the path call. Note that go places the Start state with the nil parent, [Start, nil], alone on Open_stack; Closed_set is empty:

```
go(Start, Goal) :-
    empty_stack(Empty_open),
    stack([Start, nil], Empty_open, Open_stack),
    empty_set(Closed_set),
    path(Open_stack, Closed_set, Goal).
```

The three-argument path call is:

```
path(Open_stack, _, _) :-
    empty_stack(Open_stack),
    write('No solution found with these rules').
path(Open_stack, Closed_set, Goal) :-
    stack([State, Parent], _, Open_stack), State = Goal,
    write('A Solution is Found!'), nl,
    printsolution([State, Parent], Closed_set).
path(Open_stack, Closed_set, Goal) :-
    stack([State, Parent], Rest_open_stack, Open_stack),
    get_children(State, Rest_open_stack, Closed_set, Children),
    add_list_to_stack(Children, Rest_open_stack, New_open_stack),
    union([[State, Parent]], Closed_set, New_closed_set),
    path(New_open_stack, New_closed_set, Goal), !.
get_children(State, Rest_open_stack, Closed_set, Children) :-
    bagof(Child, moves(State, Rest_open_stack,
        Closed_set, Child), Children).
moves(State, Rest_open_stack, Closed_set, [Next, State]) :-
    move(State, Next),
    not(unsafe(Next)),                          % test depends on problem
    not(member_stack([Next,_], Rest_open_stack)),
    not(member_set([Next,_], Closed_set)).
```

We assume a set of move rules, and, if necessary, an unsafe predicate:

```
move(Present_state, Next_state) :- ...                % test first rule.
move(Present_state, Next_state) :- ...               % test second rule.
....
```

The first path call terminates search when the Open_stack is empty, which means there are no more states on the open list to continue the search. This usually indicates that the graph has been exhaustively searched. The second path call terminates and prints out the solution path when the solution is found. Since the states of the graph search are maintained as [State, Parent] pairs, printsolution will go to the Closed_set and recursively rebuild the solution path. Note that the solution is printed from start to goal.

```
    printsolution([State, nil], _) :-
        write(State), nl.
    printsolution([State, Parent], Closed_set) :-
        member_set([Parent, Grandparent], Closed_set),
        printsolution([Parent, Grandparent], Closed_set),
        write(State), nl.
```

The third path call uses bagof, a PROLOG predicate standard to most interpreters. bagof lets us gather all the unifications of a pattern into a single list. The second parameter to bagof is the pattern predicate to be matched in the database. The first parameter specifies the components of the second parameter that we wish to collect. For example, we may be interested in the values bound to a single variable of a predicate. All bindings of the first parameter resulting from these matches are collected in a list and bound to the third parameter.

In this program, bagof collects the states reached by firing *all* of the enabled production rules. Of course, this is necessary to gather all descendants of a particular state so that we can add them, in proper order, to open. The second argument of bagof, a new predicate named moves, calls the move predicates to generate all the states that may be reached using the production rules. The arguments to moves are the present state, the open list, the closed set, and a variable that is the state reached by a good move. Before returning this state, moves checks that the new state, Next, is not a member of either rest_open_stack, open once the present state is removed, or closed_set. bagof calls moves and collects all the states that meet these conditions. The third argument of bagof thus represents the new states that are to be placed on the Open_stack.

In some implementations, bagof fails when no matches exist for the second argument and thus the third argument is empty. This can be remedied by substituting (bagof(X, moves(S, T, C, X), List); List = [ ]) for the current calls to bagof in the code.

Finally, because the states of the search are represented as state–parent pairs, the member check predicates, e.g., member_set, must be revised to reflect the structure of the pattern matching. We need to test to see if a state–parent pair is identical to the first element of a list of state–parent pairs and then recur if it isn't:

```
    member_set([State, Parent], [[State, Parent]| _]).
    member_set(X, [_|T]) :- member_set(X, T).
```

15.4.2 Breadth-First Search in PROLOG

We now present the *shell* of an algorithm for breadth-first search using explicit open and closed lists. The shell can be used with the move rules and unsafe predicates for any search problem. This algorithm is called by:

```
go(Start, Goal) :-
    empty_queue(Empty_open_queue),
    enqueue([Start, nil], Empty_open_queue, Open_queue),
    empty_set(Closed_set),
    path(Open_queue, Closed_set, Goal).
```

Start and Goal have their obvious values. Again we create the ordered pair [State, Parent], as we did with depth and breadth search, to keep track of each state and its parent; the Start state is represented by [Start, nil]. This will be used by printsolution to re-create the solution path from the Closed_set. The first parameter of path is the Open_queue, the second is the Closed_set, and the third is the Goal. *Don't care* variables, those whose values are not used in a clause, are written as _.

```
path(Open_queue, _, _) :-
    empty_queue(Open_queue),
    write('Graph searched, no solution found.').
path(Open_queue, Closed_set, Goal) :-
    dequeue([State, Parent], Open_queue,_), State = Goal,
    write('Solution path is: '), nl,
    printsolution([State, Parent], Closed_set).
path(Open_queue, Closed_set, Goal) :-
    dequeue([State, Parent], Open_queue, Rest_open_queue),
    get_children(State, Rest_open_queue, Closed_set, Children),
    add_list_to_queue(Children, Rest_open_queue, New_open_queue),
    union([[State, Parent]], Closed_set, New_closed_set),
    path(New_open_queue, New_closed_set, Goal), !.
get_children(State, Rest_open_queue, Closed_set, Children) :-
    bagof(Child, moves(State, Rest_open_queue,
        Closed_set, Child), Children).
moves(State, Rest_open_queue, Closed_set, [Next, State]) :-
    move(State, Next),
    not(unsafe(Next)),                              % test depends on problem
    not(member_queue([Next,_], Rest_open_queue)),
    not(member_set([Next,_], Closed_set)).
```

This algorithm is a shell in that no move rules are given. These must be supplied to fit the specific problem domain. The queue and set operators are found in Section 15.2.

The first path termination condition is defined for the case that path is called with its first argument, Open_queue, empty. This happens only when no more states in the graph remain to be searched and the solution has not been found. A solution is found in the second path predicate when the head of the open_queue and the Goal state are identical.

When path does not terminate, the bagof and moves predicates gather all the children of the current state and maintain the queue. The actions of these predicates were described in the previous section. In order to recreate the solution path, we saved each state as a state–parent pair, [State, Parent]. The start state has the parent nil. As noted in Section 15.4.1, the state–parent pair representation makes necessary a slightly more complex pattern matching in the member, moves, and print_solution predicates.

15.4.3 Best-First Search in PROLOG

Our shell for best-first search is a modification of the breadth-first algorithm in which the open queue is replaced by a priority queue, ordered by heuristic merit, for each new call to path. In our algorithm, we attach a heuristic measure permanently to each new state on open and use this measure for ordering states on open. We also retain the parent of each state. This information is used by printsolution, as in breadth-first search, to build the solution path once the goal is found.

To keep track of all required search information, each state is represented as a list of five elements: the state description, the parent of the state, an integer giving the depth in the graph of its discovery, an integer giving the heuristic measure of the state, and the integer sum of the third and fourth elements. The first and second elements are found in the usual way; the third is determined by adding one to the depth of its parent; the fourth is determined by the heuristic measure of the particular problem. The fifth element, used for ordering the states on the open_pq, is f(n) = g(n) + h(n), as presented in Chapter 4.

As before, the move rules are not specified; they are defined to fit the specific problem. The ADT operators for set and priority queue are presented in Section 15.2. heuristic, also specific to each problem, is a measure applied to each state to determine its heuristic weight, the value of the fourth parameter in its descriptive list.

This algorithm has two termination conditions and is called by:

```
go(Start, Goal) :-
    empty_set(Closed_set),
    empty_pq(Open),
    heuristic(Start, Goal, H),
    insert_pq([Start, nil, 0, H, H], Open, Open_pq),
    path(Open_pq, Closed_set, Goal).
```

nil is the parent of Start and H its heuristic evaluation. The code for best-first search is:

```
path(Open_pq, _,_) :-
    empty_pq(Open_pq),
    write('Graph searched, no solution found.').
path(Open_pq, Closed_set, Goal) :-
    dequeue_pq([State, Parent, _, _, _], Open_pq,_),
    State = Goal,
    write('The solution path is: '), nl,
    printsolution([State, Parent, _, _, _], Closed_set).
path(Open_pq, Closed_set, Goal) :-
    dequeue_pq([State, Parent, D, H, S], Open_pq, Rest_open_pq),
    get_children([State, Parent, D, H, S], Rest_open_pq, Closed_set, Children, Goal),
    insert_list_pq(Children, Rest_open_pq, New_open_pq),
    union([[State, Parent, D, H, S]], Closed_set, New_closed_set),
    path(New_open_pq, New_closed_set, Goal), !.
```

get_children is a predicate that generates all the children of State. It uses bagof and moves predicates as in the previous searches. Details are found in Section 15.4.1. Move rules, a safe check for legal moves, and a heuristic must be specifically defined for each application. The member check must be specifically designed for five element lists.

```
get_children([State, _, D, _, _], Rest_open_pq, Closed_set, Children, Goal) :-
    bagof(Child, moves([State, _, D, _, _], Rest_open_pq,
    Closed_set, Child, Goal), Children).
moves([State, _, Depth, _, _], Rest_open_pq, Closed_set,
        [Next, State, New_D, H, S], Goal) :-
    move(State, Next),
    not(unsafe(Next)),% determined by application
    not(member_pq([Next, _, _, _, _], Rest_open_pq)),
    not(member_set([Next, _, _, _, _], Closed_set)),
    New_D is Depth + 1,
    heuristic(Next, Goal, H),% determined by application
    S is New_D + H.
```

Finally, printsolution prints the solution path. It recursively finds State–Parent pairs by matching the first two elements in the state description with the first two elements of the five element lists that make up the Closed_set. The start state has nil as its parent.

```
printsolution([State, nil, _, _, _], _) :-
    write(State), nl.
printsolution([State, Parent, _, _, _], Closed_set) :-
    member_set([Parent, Grandparent, _, _, _], Closed_set),
        printsolution([Parent, Grandparent, _, _, _], Closed_set),
    write(State), nl.
```

15.5 A PROLOG Planner

In Section 6.3 we described a predicate calculus-based planning algorithm. It was predicate calculus (PC) based in that the PC representation was chosen for both the state of the planning world descriptions as well as the change of state rules. In this section we create a PROLOG version of that algorithm.

We represent the states of the world, including the begin and goal states, as lists of predicates. Two states, the start and goal states for our example, are described:

```
start = [handempty, ontable(b), ontable(c), on(a,b), clear(c), clear(a)]
goal = [handempty, ontable(a), ontable(b), on(c,b), clear(a), clear(c)]
```

These states are seen, along with a portion of the search space, in Figures 15.3 and 15.4.

The moves in this blocks world are described using *add* and *delete* lists, as in Section 8.4. The move predicates have three arguments. First is the move predicate name with its arguments. The second argument is the list of preconditions: the predicates that must be true in the description of the state of the world for the move rule to be applied to that state. The third argument is the add and delete list: the predicates that are added to and deleted from the state of the world to create the new state of the world that results from applying