their expert system and the language, or shell, in which they are written, and, in my view, one example of this kind is worth 50 pages of general description.

Let me, however, finish with some rather more positive remarks about this book. If you already understand the practical nuts and bolts of expert systems, you will certainly gain from the broader strategic overview which the author presents and will be able to compare the competitive advantage which one type of expert system might provide over another.

If you are interested in management strategy in its own right, the book contains much of value as well; for example, several sections are devoted to the management concepts involved in handling change and innovation in general, which is of considerable interest.

Delia Venables is a Fellow of the British Computer Society and a committee member of the Expert Systems Specialist Group. She has worked in systems analysis and operational research and is now an independent computer consultant.

Algorithmic methods for artificial intelligence by M Griffiths & C Palissier, Kogan Page 1986
Formal methods in artificial intelligence by J-P Delahaye, North Oxford Academic 1987

Writers and philosophers have toyed with the idea of artificial intelligence for centuries, but it was only with the advent of computers that the idea was discussed in computational terms.

Modern artificial intelligence (AI) could be said to date from Turing's work. In his *imitation game* a person would interact remotely with something and would have to guess whether it was human or machine. If the person was not able to tell the difference, then the machine could be said to be intelligent. The aim of the researcher and implementor of such a machine is to discover what humans can discuss, and then emulate this activity. This leads to the direction of research which could be described as a replacement strategy. For example, early work in the more specialized field of expert systems attempted this approach—to replace experts. It also implies that the machine will be indistinguishable from a human and hence that we must assume that humans can only do that which is done by computer. For whatever reason, computer models are now used to test theories of the mind—a large part of the work of people in cognitive studies.

Turing's test can also be interpreted as an injunction to design and build something and only then discuss what properties of intelligence it has, rather than merely sitting around philosophizing about intelligence. In this way it fits more into our daily work patterns. Most computing professionals, whether academic or not, whether working in AI or not, are involved in creating usable and useful systems. To do this we need to explore what computers can do and what humans can do, and create systems which appropriately mesh these two components. The aim is hopefully, to enhance human lives.

From this second viewpoint, Turing and Church's earlier mathematical work is most important. For they discussed the mathematical constraint on computers. This is expressed succinctly in Delahaye's book:

> The only problems that a computer program is able to solve are the *decidable* problems, that is, problems that can be expressed as a decidable predicate; and this term has a strict interpretation that is independent of the programming language.

Delahaye takes us along the path followed by older courses in logic and computation. Through decidability and recursive functions to formal systems and then propositional and predicate logic, that is, logics that can be used to model collections of simple statements. Thus "only criminals commit crimes" would be written as $\forall\, y\, \forall\, x\, ((cr(x) \wedge co(y,x) \rightarrow cm(y))$, where cr(x) is an abbreviation for "x is a criminal", co(y,x) is an abbreviation for "x commits y" and cm(y) for "y is a crime". There are several important issues being discussed here. Firstly, notation used for abbreviating logical statements whether used formally or informally, the intuitions that a particular logic system is trying to represent, the syntax of a particular formalization, the semantics of that formalization

and whether that formalization can be embedded in a computer system which is able to decide whether a particular statement in a particular context follows from the known facts of that context. Formal logical systems are useful even when those systems are not *decidable*. For this reason more people are now teaching logic separately from decidability.

It is good to know that predicate calculus as in the example above is not decidable, and that no adequate formal system for arithmetic is decidable. Hopefully it will stop those rash people who make statements such as "in theory, I could model the whole human mind". We can't even write a program which will take any statement about arithmetic and state whether it is true (Hofstadter, 1980). Nor can we write a program which can take any predicate logic expression and state whether or not it is a theorem of first order predicate logic. However, we can design computers and programs not only to do arithmetic, but also quite complicated things with logical systems—for, after all a formal logical system is a collection of symbols with well-defined manipulation rules, and computers can be thought of as symbol manipulators.

In a lot of AI work we need some sort of *inference engine* which can be given facts, rules and queries and draw the appropriate inferences. Whilst this can be programmed in any programming language it is arguably better done in a programming language which is designed to represent logical reasoning. Thus was born the field referred to as logic programming and in particular the logic programming language prolog. A logic program consists of little more than facts such as "Keith committed fraud", "fraud is a crime" and the rule mentioned above "only criminals commit crimes". Using the abbreviations above we could write the first two as co(Keith,fraud) and cr(fraud). The rule above is written with implicit $\forall$ phrases, that is, as $((cr(x) \wedge (co(y,x) \rightarrow cm(y))$, with x,y as variables. This rule can be read two ways: "if cr(x) and co(y,x) are true then we can conclude that cm(y) is true" or alternatively as "to show that cm(y) is true it is necessary and sufficient to show that cr(x) and co(y,x) are true". Thus, it can be used in either direction, and should be thought of as a known theorem. The basis of logic programming languages is the means whereby we can bring together such symbolic statements. There are two aspects: *unification* which in the above links x with fraud and y with Keith and *resolution* which brings together the statements to draw the appropriate conclusion with the appropriate substitutions, which in the above is cm(Keith). Delahaye gives a thorough explanation, with good examples of various algorithms for carrying out unification and resolution. Such algorithms have great practical utility even though they do not and can not, provide a general program for testing whether any statement is a theorem of predicate logic. Actual logic programming languages have other restrictions about the allowed form and syntax of statements, but the principles are the same.

It is convenient to think of a lot of problems in AI as problems of search or planning. Most of these problems are too complex to allow for complete enumeration of possibilities and so it becomes necessary to formulate efficient algorithms to obtain the path to a goal state. The algorithms described in Delahaye are some of those used to find a deduction path for a theorem.

Griffiths and Palissier briefly describe the reasoning behind backtracking, minimax, learning algorithms and alpha-beta pruning, as well as a couple of algorithms for theorem proving. Their book is part of Kogan Page's New Technology Modular Series which includes other books well worth a read e.g. Addis, 1985; Alexander, 1986. Most of the Griffiths and Palissier book is devoted to setting the context of the algorithms and contains chapters on the history of AI, theorem proving, expert systems, data structures, algorithms and programming languages for AI. It is much more introductory than some of the other books in the series and is more suited to a reader with no knowledge of AI and only a little knowledge of programming—it could be also used as a text for a single AI course following on from an introductory programming course. However, it unfortunately does not contain very much about algorithmic methods for AI! I do not think it is suitable for use in a full scale computer science or AI programme except as supplementary reading, as there are better ways to construct such a programme.

Formal systems and logic in particular are useful for several aspects of computer science and AI—not only for logic programming and theorem provers, but also for formal specification methods, modern programming languages and databases. As such, logic needs to be introduced

early in the curriculum using an easy to read book such as Hodges, 1985, and then followed up more mathematically in a course on nonmonotonic and temporal logics, logics of belief, etc. and in a course on logic programming. Computability can be handled either in a course of this type or in a course on complexity. Algorithms and data structures for AI could form another whole course.

Just as you can hammer in a nail with the handle of a screwdriver, so you can use C or Pascal to write AI programs. However it is always easier and better to use an appropriate tool for the task. Various programming languages have been designed with AI applications in mind: Lisp, prolog, popII, ops5 to name the main contenders. Similarly, object oriented languages have been used to write real-time expert systems and blackboard systems. The AI practitioner needs to understand the philosophy, not only of older procedural languages, but also the paradigms of functional programming, list processing languages, logic programming and object-oriented programming. It can be argued that these paradigms are more important than the languages. Lisp was originally designed as a functional list-processing language and versions are available with object-oriented extensions—it therefore combines most of the features of these paradigms. The poplog environment developed at Sussex contains procedural and logic capabilities as well as the functional list processing language popII. Newer versions of prolog contain support for functions and object-oriented extensions have been written. Modern functional languages such as ML and Miranda use unification techniques in a very small way.

Delahaye briefly introduces prolog II developed at Marseille as the successor of prolog. Griffiths and Palissier introduce both Lisp and prolog II—although all the algorithms in the book are written in a language-free procedural form. One should note that they introduce the languages rather than the paradigms, and use the Marseille notation for prolog rather than the notation that is commonly referred to as Edinburgh prolog. One of the most unfortunate features of this is the use of A → B to mean that B entails A rather than the reverse.

In logic programming it is important to think in terms of theorems and proofs rather than in operational semantics. Ideally the code should reflect this. Too often a person moving from a procedural or functional language to logic programming writes code that is not *backwards correct*—that is it only works in one direction or is not *steadfast*—that is, it is not valid in all contexts. Consider the following piece of code from Delahaye:

```
conc2(x.y,x.z,t) - >conc2(y,z,t) ;

conc2(x,nil,x) - >;

inv1(nil,nil) - >/ ;

inv1(z,x.y) - >inv1(w,y) conc2(z,w,x.nil) ;
```

Disregard the names and questions of efficiency. The period (.) is the list constructor, i.e. x.y is the list with head x and tail y. Thus the first line says that x.y is the concatenation of x.z and t if y is the concatenation of z and t and the second line that x is the concatenation of the empty list and x.conc2 works both ways, whichever arguments are partly or fully instantiated, the appropriate instantiations are made if possible and if not possible we are told that it is not. Thus if one asks whether there is an instantiation of m,n such that conc2(A.m,A.B.C.nil,D.n) is true, it returns the expected answer that it is true when m and n are of the form B.C.D.w and w respectively. A useful predicate is inv(x,y) to mean that x is the reverse of y. However, inv1 is only a partial implementation of list reversal. It can only be used to instantiate x to the result of reversing y or to say no if x but not y is instantiated in the top level goal. The slash used in inv1 is the much abused *cut* of logic programming implementations and is rapidly acquiring the same status as the goto statement in procedural languages. It is easy enough to write a decent list reversal predicate in prolog (see Bratko, 1986). It is generally preferable to have the base case of predicates first, e.g. conc2(x,nil,x); at least one should be consistent. Delahaye is aware of the limitations of his code and it is unfortunate that it is included in an introductory book as it is better used as an example of bad logic programming.

Current prolog systems contain features that cause infinite looping in otherwise correct logic programs. A prolog programmer should be aware of these but avoid bad programming habits by planning predicates first in pure logic and then in prolog so that they can be read as theorems—both steadfast and backwards correct. With the advent of parallel treatment of individual predicates in a clause, non-deterministic matching of heads of clauses and more high level control structures, the current problems will largely disappear and pure logic programming will not need the low level devices used in the code here.

Griffiths and Palissier is a small, easy to read introduction to AI for those with no former knowledge, and Delahaye a thorough introduction to decidability in the context of formal systems suitable for those with reasonable mathematical ability. However, I would advise everyone not knowledgeable about logic to seek this knowledge elsewhere. There are several books on current AI programming languages (Bratko, 1986; Winston and Horn, 1981) which will expand the reader's knowledge of AI as well as the language. These would be a better introduction than the somewhat unsatisfactory coverage in the two reviewed books.

Dave Saunders of Dept of computer science and statistics, Queen Mary College, London E1 4HS.

References

Hofstadter, D R, 1980. *Godal, Escher, Bach: An Eternal Golden Braid*, Vintage Books.
Addis, T R, 1985. *Designing Knowledge-Based Systems*, Kogan Page.
Alexander, I, 1986. *Designing Intelligent Systems*, Kogan Page.
Hodges, W, 1985. *LOGIC*, Pelican.
Bratko, I, 1986. *PROLOG programming for Artificial Intelligence*, Addison-Wesley.
Winston, P H and Horn, B K P, 1981. *LISP*, Addison-Wesley.