THE EARLY YEARS OF LOGIC PROGRAMMING

This firsthand recollection of those early days of logic programming traces the shared influences and inspirations that connected Edinburgh, Scotland, and Marseilles, France.

ROBERT A. KOWALSKI

The name *Prolog* is ambiguous. It was originally intended as the name for the programming language developed by Alain Colmerauer and Phillipe Roussel in the summer of 1972. The name was suggested by Roussel's wife, Jacqueline, as an abbreviation for pro*grammation en* log*ique*. In time, however, this abbreviation has been used to refer to the concept of logic programming in general. It is a confusing notion, as claims made for the general concept of logic programming do not always hold for the programming language, Prolog, and vice versa. In an attempt to minimize such confusion, I shall reserve the term Prolog to refer to the programming language alone.

This is not the place for an extensive discussion of what should or should not be regarded as *logic programming*, a term that is equally ambiguous. However, without wanting to stir further controversy, let me hazard the following rough characterization: Logic programming shares with mechanical theorem proving the use of logic to represent knowledge and the use of deduction to solve problems by deriving logical consequences. However, it differs from mechanical theorem proving in two distinct but complementary ways: (1) It exploits the fact that logic can be used to express definitions of computable functions and procedures; and (2) it exploits the use of proof procedures that perform deductions in a goal-directed manner, to run such definitions as programs.

A consequence of using logic to represent knowledge is that such knowledge can be understood *declaratively*. A consequence of using deduction to derive consequences in a computational manner is that the same knowledge can also be understood *procedurally*. Thus, logic programming allows us to view the same knowledge both declaratively *and* procedurally.

The most straightforward case of logic programming is when information is expressed by means of Horn clauses and deduction is performed by backwards reasoning embedded in resolution [29]. But logic programming can also be understood more generally, for example, to include negation by failure [3], set construction [4, 32], or goal-directed reasoning with equations. The advantage of the more liberal notion of logic programming is that it points the way for further developments to encompass richer fragments of logic and give a computational interpretation to a greater variety of proof procedures.

The liberal notion of logic programming does not include a number of related uses of logic in programming. It excludes, for example, systems of constructive logic in which *proofs* are interpreted as programs, and it excludes uses of logic in which computation is construed model-theoretically as evaluating a formula in an interpretation.

This article is a personal account of some of the early history of logic programming, ending with my move from Edinburgh to London in December 1974. The chronicle is unavoidably biased toward my own recollection of events at the University of Edinburgh. I am especially conscious that it does not do justice to related activities that took place during that time at the Université d'Aix Marseilles.

THE EDINBURGH–MARSEILLES CONNECTION

My first contact with the Marseilles group was a three or four day visit in the summer of 1971 at the invitation of Colmerauer, who was then head of the artificial intelligence (AI) team at the university. The group, which consisted of Bob Pasero, Roussel, and Colmerauer, was developing a natural language question-answering system. Roussel and Jean Trudel, a colleague visiting from the University of Montreal, had read [21], which describes the SL-resolution theorem prover, and Roussel was interested in using it for the deductive component of the question-answering system.

Most of my visit consisted of intensive discussions

© 1988 ACM 0001-0782/88/0100-0038 $1.50

with Colmerauer about using logic to represent grammar and using resolution to parse sentences. Earlier, I had devised an inefficient representation of grammars, with explicit axioms of associativity for string concatenation. Colmerauer saw how to improve the representation significantly, avoiding associativity by formalizing the graph representation of strings used in his Q-Grammars [5]. We observed that the bottom-up behavior of his Q-system parser could be obtained by using hyperresolution. SL-resolution behaved as a top-down parser. It is because of this work that 1971 is sometimes given as the year Prolog was born.

My short visit was very productive, and we planned to continue our collaboration. Our plans were realized in the spring of 1972 during my second visit to Marseilles. My trip was again at Colmerauer's invitation. This time I was accompanied by doctoral student, Ed Wilson.

During this period the idea of programming in predicate logic was born. I had been asked to serve as external examiner for Roussel's *Thèse de Troisième Cycle* [30]. I was impressed by his use of "formal equality" (characterized by the single axiom $x = x$) to avoid the inefficiencies of the normal equality axioms in certain applications. This lead me to look for other cases where a change of representation could lead to improved efficiency. It was not long before I could see how to write computationally efficient axioms for such recursive predicates as addition and factorial. With Ed Wilson and Roussel, I looked at both Horn clause and non-Horn clause definitions "executed" by SL-resolution. Roussel, in turn, spoke with Colmerauer and reported back ideas that arose during their conversations. I did not realize until much later how closely Colmerauer's work paralleled my own.

Colmerauer and I had quite different backgrounds and placed different values on different things. Colmerauer was a computer scientist who combined practical achievements with sound contributions to their theory. I was a logician at heart, who suffered a faint revulsion for programming and everything else to do with computers. As a student, I loved logic and hated recursion theory.

Looking back on our early discoveries, I value most the discovery that computation could be subsumed by deduction. Colmerauer was not so readily satisfied with purely theoretical results. For him, the Horn clause definition of appending lists was much more characteristic of the importance of logic programming: It provides a basis for more powerful programming methods and is ideally suited to nonnumerical applications such as natural language processing. This difference of emphasis, which still exists today, may reflect the difference between our early contributions to the subject.

From Marseilles I wrote to Bernard Meltzer at the University of Edinburgh to explain the new ideas. In his reply, Meltzer wrote that my letter "generated a lot of discussions." Pat Hayes, in particular, argued that I seemed to be taking credit for the thesis that "computation is controlled deduction," which he had been advocating in Edinburgh before me.

Indeed, Hayes had argued that computation and deduction were similar some time before my second visit to Marselles. In particular, he argued that inference with equations imitated computation in Lisp and that Robert Boyer and J Moore's new structure-sharing method of implementing resolution [1] gave similar run-time structures to the Bobrow and Wegbreit spaghetti stack mechanism. Hayes received little credit for his ideas, and to a large extent, I failed to appreciate his ideas both because I had never programmed in Lisp and because I did not take much interest in implementation.

Hayes had been my closest friend at the University of Edinburgh since my arrival as a Ph.D. student in October 1967. The first research either one of us did was a combined effort that resulted in a paper on semantic trees in *Machine Intelligence*, vol. 4. We were collaborating on a book on automated theorem proving and had finished a substantial part of it before Hayes left Edinburgh for a second visit to Stanford University. At Stanford he learned about Planner [12], and when he returned to Edinburgh, he wanted to rewrite our book significantly to take Planner into account. We spent many hours discussing and arguing the relationship between Planner and resolution theorem proving. These discussions were part of the background to my second visit to Marseilles. A few months after I returned, Hayes left Edinburgh for a lectureship at the University of Essex.

Let me hazard the following rough characterization: Logic programming shares with mechanical theorem proving the use of logic to represent knowledge and the use of deduction to solve problems by deriving logical consequences.

When I returned from Marseilles, Boyer and Moore were very enthusiastic. Programming in resolution logic seemed to be just what they were looking for to exploit their earlier discovery of the structure sharing method of implementing resolution. They were already aware that structure sharing was analogous to the use of association lists in the implementation of Lisp. By the summer of 1972, however, they were so enthusiastic that they developed their own logic programming language called Baroque [27].

Baroque was an assembly-like programming language that provided list processing and arithmetic primitives defined by Horn clauses and interpreted by a structure-

sharing SL-resolution theorem prover with a depth-first search strategy. "Demons" were attached to primitive functions such as addition and multiplication, to exploit the machine's built-in arithmetic. Boyer and Moore then coded an interpreter in Baroque for a programming language akin to pure Lisp, with pattern-matched invocation and nondeterminism inherited from its implementation language. Programs written in the Lisp-like language were about 10 times slower than similar programs written directly in Baroque. However, what intrigued Boyer and Moore about the language was that it permitted (indeed, encouraged) the symbolic execution of programs and hence the interpreter could prove simple theorems about programs, such as "There exists an X such that `(Length X)` is 3." As they tried to prove more complicated theorems about programs, for example, that `append` is associative or that the `length` of `(Append` *A B*`)` is the sum of the `lengths` of *A* and *B*, they realized it was necessary to do mathematical induction. Boyer and Moore then turned their attention toward mechanizing inductive proofs [2].

In the eyes of most North American researchers in AI, resolution had long been discredited. The fashion had turned . . . toward the procedural representation of knowledge and domain-specific problem solvers.

The summer of 1972 was a busy time for the development of logic programming. Back in Marseilles, Roussel and Colmerauer designed and implemented the first Prolog system in Algol-W as an adaptation of Roussel's existing SL-resolution theorem prover. Teaming with Pasero they implemented a large natural language processing system. This was the first major program written in Prolog [6], and it was written in 1972. I continued my own investigations and reported my findings [15] at the first "Mathematical Foundations of Computer Science" conference in Jablonna, Poland, in August 1972.

The situation in Edinburgh during the next year was stormy, to say the least. In the eyes of most North American researchers in AI, resolution had long been discredited. The fashion had turned against uniform, general-purpose theorem provers toward the procedural representation of knowledge and domain-specific problem solvers.

Those of us in Edinburgh who continued working in the resolution paradigm were increasingly isolated from the rest. I was fortunate, however, to be joined in my work by David Warren and Maarten van Emden. During this time our contact with Marseilles was a great inspiration and comfort. I wrote an application for a NATO research grant to fund exchanges between our two groups to investigate further the application of logic programming to natural language processing. Despite some last-minute problems with a hostile potential referee, the grant was finally approved. It supported several visits during the period of October 1973 through September 1974 by Warren, Steve Isard, Bob Welham, van Emden, and myself to Marseilles. It also allowed Colmerauer, Roussel, and Henri Kanoui to visit Edinburgh. By March 1974, I had completed a 100-page draft of "Logic for Problem Solving" [18]. This was widely circulated for many years before the expanded version [20] was published in 1979.

I was an avid supporter of coroutining for Horn clause logic programs during this period. I described this in [16] at the "IFIP 74" conference in Stockholm. I had many discussions about this with Roussel and encouraged Robert Hill, a Ph.D. student of Meltzer's who was informally under my supervision, to investigate its properties. He invented the name *Lush* (*l*inear resolution with *u*nrestricted *s*election for *h*orn clauses) and proved its completeness in [13]. Krystof Apt and van Emden later used the now more familiar term *SLD-resolution* for the same system.

Among Warren's many interesting studies before visiting Marseilles was his investigation of programming with non-Horn clauses. He interpreted this as the analog of block structure in Algol-like languages. Of more lasting significance, however, was his conversion to Prolog during his visit to Marseilles. I can recall his indignation when he returned over my lack of support for Prolog. Certainly, given my previous research in heuristic search [14], I found it hard to accept Prolog's incomplete depth-first, backtracking search strategy. Moreover, with my theorem proving background, I also found it hard to be enthusiastic about Prolog's sequential execution of procedure calls. I hoped it might be possible to base a more powerful logic programming system on the use of coroutining in Lush or in the connection graph proof procedure [17]. But Prolog was a practical programming language, whereas at that time Lush and connection graphs were not. It was not until around 1976 when I was at Imperial College in London that I finally appreciated the ingenious, delicate balance that Prolog achieved between being a fairly primitive, but useful, theorem prover, and being a very high-level programming language.

In 1973 I worked with van Emden investigating the relationship between Scott's fixed point semantics of recursive programs and the Tarskian semantics and proof theory of first-order logic [31]. We also had broader aspirations of adapting existing techniques for proving properties of recursive functions, such as Scott's fixed point induction, to logic programs. But we were unable to do so within the time constraint we set

for ourselves for the semantics paper. These broader problems were later solved by Keith Clark at Imperial College.

Most of the early converts and contributors to logic programming became so as the result of personal contacts and discussions rather than through industry publications. In addition to the many I have mentioned are Luis Moniz Pereira's group in Lisbon, Portugal, the Hungarian Prolog activities, and the work of Sten-Åke Tärnlund's group in Sweden. All their efforts were similarly initiated as the result of contacts with the Edinburgh team.

Luis Moniz Pereira came to the University of Edinburgh to work as a research fellow in 1974–1975. He was an active contributor to a working group we organized to develop logic programs for geometry theorem proving. He also contributed to Warren's Prolog compiler and later, when he returned to Lisbon, worked with Fernando Pereira until 1978.

Istvan Nemeti arrived at Edinburgh the day I left for Imperial College in December 1974. He collaborated with van Emden on semantic issues of logic programming and took back to Hungary a copy of Warren's notes on Prolog implementation. These notes were the beginning of Prolog in Hungary.

I met Tärnlund at the IFIP conference in August 1974. He had already implemented an SL-resolution theorem prover and was planning to implement a connection graph theorem prover. He was immediately attracted to logic programming. I visited him for two weeks in Stockholm in December 1974.

I left Edinburgh for a readership at Imperial College in December 1974, leaving my work with Warren and van Emden. At Imperial College there was no previous interest in theorem proving or logic programming. I was very fortunate that Clark was able to take a two-year leave of absence from Queen Mary College to work with me on an SERC grant. Today, the Logic Programming Group at Imperial College includes 2 professors 1 reader, 7 lecturers, 3 SERC advanced research fellows, 19 research assistants, 5 administrative and clerical staffers, and 13 research students, for a total of 50.

Most of the early converts and contributors to logic programming became so as the result of personal contacts and discussions rather than through industry publications.

THE RELATIONSHIP BETWEEN LOGIC PROGRAMMING, SL-RESOLUTION, AND PLANNER

Perhaps the first zenith of logic in AI occurred when Cordell Green illustrated how to represent question-answering, plan formation, program synthesis, and program simulation in first-order logic [10]. His attempts to use resolution for problem solving in these domains, however, were less successful. The resolution systems that had been developed by that time and were at his disposal were intolerably redundant, combinatorially explosive, and unnatural in behavior. The early enthusiasm that greeted Green's work soon gave way to a massive backlash. Terry Winograd's thesis probably offered the most eloquent and influential voice to the attack. The alternative, which he and others advocated, was a procedural rather than a declarative representation of knowledge and the employment of domain-dependent problem solvers rather than uniform, general-purpose theorem provers.

Carl Hewitt's programming language Planner [12] was regarded as the embodiment of these ideas. However, Planner was also based on logic. As Hewitt explained, an implication "*A if B*" could be interpreted as four different Planner procedures:

(1) To show *A*, show *B*;
(2) to show not-*B*, show not-*A*;
(3) given *B*, assert *A*; and
(4) given not-*A*, assert not-*B*.

Linear resolution could implement the first two of these; and hyperresolution with renaming, the second two. But the early versions of these methods were highly redundant. Linear resolution in particular, given an implication

A if B_1 and B_2 and ... and B_n,

would attempt to solve the subgoals $B_1, \ldots, B_n$ in all n factorial ways. It was ironic that Loveland [23], who with Luckham [24] independently invented linear resolution, did not at first see the connection with his own model elimination proof procedure [22]. Model elimination, using a very different formalism, could be interpreted as a form of linear resolution without the n factorial redundancy. Eventually, the connection between the two was independently noted both by Loveland and by Kuehner and myself. We called the resulting synthesis SL-resolution (linear resolution with selection function). At the same time, Reiter [28] discovered the same ordering restriction on linear resolution. Virtually all obvious redundancies were removed from linear resolution by this stage, and most of Green's examples could have been rerun with much greater success.

I can recall trying to convince Hewitt that Planner was similar to SL-resolution. Planner gave a problem reduction interpretation to logic in an ad hoc but pragmatic fashion. It gave the programmer greater control than general, unrestricted resolution, but was also less uniform. Its pattern matcher, in particular, was both more complicated and more restricted than full unification. Moreover, because of the way it was embedded in Lisp, it was not easy to determine whether Planner without Lisp was itself a general-purpose programming language. As Colmerauer once said, no one would try to define the list `append` operation in pure Planner without resorting to Lisp.

This last reservation about Planner as a programming

language also applied to SL-resolution. Although the problem-reduction behavior of SL-resolution was appreciated from the very beginning, it was not clear at first that this was sufficient to make clausal logic or the Horn clause subset a general-purpose programming language. It was this realization that we now associate with logic programming and Prolog. Moreover, it was this idea that was missed when the early critics of Prolog mistakenly regarded it simply as a reinvention of Planner.

THE RELATIONSHIP BETWEEN LOGIC PROGRAMMING AND COMPUTATION = CONTROLLED DEDUCTION

The relationship of logic programming with Hayes's Golux idea [11] that *computation = controlled deduction* is both subtle and complex. On the one hand, Golux is more general. Hayes first applied his notion to Lisp, before the discovery of logic programming, whereas in [11] he applied it to programs expressed by means of equations. On the other hand, Golux is also more specific. Given a description L of a problem formulated in logic, Hayes advocated varying the control C of the problem solver/theorem prover to obtain efficient, computational behavior. He argued against altering the declarative component L to make it easier to control.

From a logic programming point of view, the Golux idea is more like running programming specifications than it is like writing programs.

From a logic programming point of view, the Golux idea is more like running program specifications than it is like writing programs. It is central to the idea of logic programming, and of Prolog in particular, that we be prepared to alter the declarative component L of a problem description to obtain desired problem-solving behavior A from a given control C. This has been expressed by the pseudoequation [19]

$$A(lgorithm) = L(ogic) + C(ontrol).$$

With Prolog we have a fixed C and can improve A only by improving L. Hayes, however, emphasized the value of changing A by changing only C. Logic programming is concerned with the possibility of changing both L and C.

Golux was influenced by Absys, a declarative programming language developed at the University of Aberdeen and reported in a number of papers in the *Machine Intelligence* series [7–9]. Absys anticipated a number of Prolog features, such as "invertability," "negation by failure," "aggregation operators," and the central role of backtracking. Like Golux it emphasized the separation of logic from control and the value of changing A by changing C.

McCarthy's interpretation of logic programming, applied to the map coloring problem [25], fixes the logic of the problem and attempts to obtain a desired algorithmic behavior by changing only the control. This is in the Golux spirit.

McDermott's recent criticism of the "logicist" position on the role of logic in AI [26] seems to be addressed primarily to the Golux idea. He advocates a return to procedural representations of knowledge, while acknowledging that representations that have both denotational (logic) and procedural semantics would be ideal.

Acknowledgments. I am grateful to Bob Boyer, Frank Brown, Alan Bundy, Alain Colmerauer, Ted Elcock, Pat Hayes, Donald Loveland, David Luckham, John McCarthy, J Moore, Alan Robinson, David Warren, and Bob Welham for their helpful comments on earlier drafts of this paper.

REFERENCES

1. Boyer, R.S., and Moore, JS. The sharing of structure in theorem proving programs. In *Machine Intelligence*, vol. 7, B. Meltzer and D. Michie, Eds. Edinburgh University Press, Edinburgh, U.K., pp. 101–116.
2. Boyer, R.S., and Moore, JS. Proving theorems about LISP functions. *J. ACM 22*, 1 (Jan. 1975), 129–144.
3. Clark, K.L. Negation as failure. In *Logic and Data Bases*, H. Gallaire and J. Minker, Eds. Plenum, New York, 1978, pp. 293–322.
4. Clark, K.L., McCabe, F.G., and Gregory, S. IC-PROLOG language features. In *Logic Programming*, K.L. Clark and S.-Å. Tärnlund, Eds. Academic Press, New York, pp. 253–266.
5. Colmerauer, A. Les Systèmes-Q ou un formalisme pour analyser et synthétiser des phrases sur ordinateur. Rep. 43, Dept. of Computer Science, Univ. of Montreal, Quebec, 1970.
6. Colmerauer, A., Kanoui, H., Pasero, R., and Roussel, P. Un système de communication homme–machine en Français. Rep., Groupe d'Intelligence Artificielle, Univ. d'Aix Marseille II, Luminy, France, 1973.
7. Elcock, E.W. Descriptions. In *Machine Intelligence*, vol. 3, D. Michie, Ed. Oliver and Boyd, Edinburgh, U.K., 1968, pp. 173–180.
8. Foster, J.J., and Elcock, E.W. Absys 1: An incremental compiler for assertions—An introduction. In *Machine Intelligence*, vol. 4, D. Michie, Ed. Edinburgh University Press, Edinburgh, U.K., pp. 423–429.
9. Foster, J.M. Assertions: Programs written without specifying unnecessary order. In *Machine Intelligence*, vol. 3, D. Michie, Ed. Edinburgh University Press, Edinburgh, U.K., 1968, pp. 387–392.
10. Green, C.C. Application of theorem-proving to problem solving. In *Proceedings of IJCAI-1*, D.E. Walker and L.M. Norton, Eds. (Washington, D.C.). IJCAI, 1969, pp. 219–240.
11. Hayes, P.J. Computation and deduction. In *Proceedings of the 2nd MFCS Symposium*. Czechoslovak Academy of Sciences, 1973, pp. 105–118.
12. Hewitt, C. PLANNER: A language for proving theorems in robots. In *Proceedings of IJCAI-1* (Washington, D.C.). IJCAI, 1969, pp. 295–301.
13. Hill, R. LUSH resolution and its completeness. DCL Memo 78, School of Artificial Intelligence, Univ. of Edinburgh, U.K., Aug. 1974.
14. Kowalski, R.A. Search strategies for theorem proving. In *Machine Intelligence*, vol. 5, B. Meltzer and D. Michie, Eds. Edinburgh University Press, Edinburgh, U.K., 1969, pp. 181–201.
15. Kowalski, R.A. The predicate calculus as a programming language. In *Proceedings of the International Symposium and Summer School on Mathematical Foundations of Computer Science* (Jablonna, Poland, Aug.). 1972.
16. Kowalski, R.A. Predicate logic as a programming language. DCL Memo 70, School of Artificial Intelligence, Univ. of Edinburgh, U.K., Nov. 1973. (Also in *Proceedings of IFIP 1974* (Stockholm, Sweden). North-Holland, Amsterdam, 1974, pp. 569–574).
17. Kowalski, R.A. A proof procedure using connection graphs. DCL Memo 74, School of Artificial Intelligence, Univ. of Edinburgh, U.K., 1973. (Also: *J. ACM 22*, 4 (Oct. 1974), 572–595.)

18. Kowalski, R.A. Logic for problem solving. DCL Memo 75, School of Artificial Intelligence, Univ. of Edinburgh, U.K., 1974.
19. Kowalski, R.A. Algorithm = logic + control. *Commun. ACM 22*, 7 (July 1979), 424–436.
20. Kowalski, R.A. Logic for problem solving. Elsevier North Holland Inc., New York, 1979.
21. Kowalski, R.A., and Kuehner, D. Linear resolution with selection function. School of Artificial Intelligence, DCL Memo 34, Univ. of Edinburgh, U.K., 1971. (Also: *Artif. Intell.* 2 (1971), 227–260.)
22. Loveland, D.W. Mechanical theorem-proving by model elimination. *J. ACM 15*, 2 (Apr. 1968), 236–251.
23. Loveland, D.W. A linear format for resolution. In *Proceedings of the INRIA Symposium on Automatic Demonstration.* Springer-Verlag, New York, 1970, pp. 147–162.
24. Luckham, D. Refinement theorems in resolution theory. In *Proceedings of the INRIA Symposium on Automatic Demonstration.* Springer-Verlag, New York, 1970, pp. 163–190. December 1968, Rocquencourt, France.
25. McCarthy, J. Coloring maps and the Kowalski doctrine. 1982. Unpublished. Stanford University.
26. McDermott, D. Critique of pure reason. *Comput. Intell.* To be published.
27. Moore, J. Computational logic: Structure sharing and proof of program properties, Parts I and II. DCL Memo 67, School of Artificial Intelligence, Univ. of Edinburgh, Edinburgh, U.K., 1974.
28. Reiter, R. Two results on ordering for resolution with merging and linear format. *J. ACM 18*, 4 (Oct. 1971), 630–646.
29. Robinson, J.A. A machine-oriented logic based on the resolution principle. *J. ACM 12*, 1 (Jan. 1965), 23–41.
30. Roussel, P. Definition et traitement de l'egalité formelle en démonstration automatique. Thesis, Faculté des Sciences, Univ. d'Aix-Marseille, Luminy, France, 1972.
31. van Emden, M.H., and Kowalski, R.A. The semantics of predicate logic as a programming language. *J. ACM 23*, 4 (Oct. 1976), 733–742. (Also: DCL Memo 73, School of Artificial Intelligence, Univ. of Edinburgh, U.K., Feb. 1974.)
32. Warren, D.H.D. Higher-order extensions to PROLOG: Are they needed? In *Machine Intelligence*, vol. 10, J.E. Hayes and D. Michie, Eds. Wiley, New York, 1982, pp. 441–454.

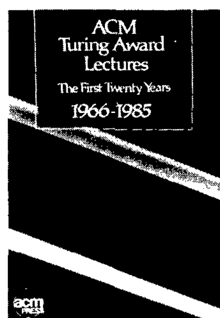
CR Categories and Subjects Descriptors: D.3.2 [**Programming Languages**]: Language Classifications—*Prolog*; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*logic programming*; I.2.3 [**Artificial Intelligence**]: Deduction and Theorem-Proving—*logic programming*; K.2 [**Computing Milieux**]: History of Computing—*people*; *software*; *systems*; *theory*
General Terms: Languages, Theory
Additional Key Words and Phrases: Absys, Baroque, Golux, linear resolution, Lush, model elimination, Planner, resolution, SL-resolution, SLO-resolution, structure sharing

Author's Present Address: Robert A. Kowalski, Imperial College of Science and Technology, University of London, 180 Queen's Gate, London, SW7 2BZ England.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

Perlis · Wilkes · Hamming · Minsky Wilkinson · McCarthy · Dijkstra Bachman · Knuth · Newell · Simon Rabin · Scott · Backus · Floyd Iverson · Hoare · Codd · Cook Thompson · Ritchie · Wirth Karp

10% savings for ACM members!



What do these prominent computer scientists have in common?

They're all recipients of the ACM's revered Turing Award, and their unique insights have been compiled for you in **ACM Turing Award Lectures: The First Twenty Years: 1966–1985.** It's the first book in the **Anthology Series** from the newly formed **ACM Press** (a unique collaboration between ACM and Addison-Wesley Publishing Company).

After introductions from Robert Ashenhurst (Anthology Series editor) and Susan Graham, **ACM Turing Award Lectures** presents the provocative lectures delivered by the 23 Turing Award recipients. Several of the recipients have, in addition, written postscripts for their lectures to give you an updated perspective on significant changes in the field.

Fuel for creativity.

These 23 masters and innovators of computer science will broaden your horizons and inspire your own creative efforts in the field. So don't miss out on the opportunity to add this enlightening book to your professional collection at the special ACM members' price of $31.50. Order today!

YES! I want to take advantage of the special 10% savings for ACM members. Please send me **ACM Turing Award Lectures: The First Twenty Years: 1966–1985** (0-201-07794-9) at $31.50 (nonmembers may purchase for the regular price of $34.95).

I've enclosed a check for $ ________, the total of my order plus $3.00 for postage and handling.

ACM Membership # ________

I prefer to charge my order. I understand I'll be charged for shipping and handling.

☐ VISA ☐ MasterCard (Interbank # ________)
☐ American Express

Account # ________ Exp. Date ________

Signature ________

Please ship ☐ UPS ☐ U.S. Mail

Name ________

Address ________

City ________, State ________ Zip ________

Send order with payment to: **ACM Order Department** · PO Box 64145 · Baltimore, MD 21264

Or call toll free 1-800-342-6626. In Alaska, Maryland, and outside the U.S., call 301-528-4261.

Addison-Wesley Publishing Company
We publish the leaders.

16018