

STANFORD ARTIFICIAL INTELLIGENCE PROJECT
MEMO AIM-160

COMPUTER SCIENCE DEPARTMENT
REPORT NO. CS-255

AD 740140

AUTOMATIC PROGRAMMING

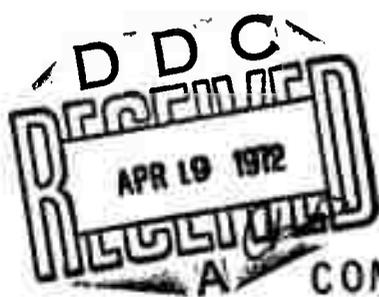
BY

JEROME A. FELDMAN

SPONSORED BY

NATIONAL SCIENCE FOUNDATION
AND
ADVANCED RESEARCH PROJECTS AGENCY
ARPA ORDER NO. 457

FEBRUARY 1972



COMPUTER SCIENCE DEPARTMENT

STANFORD UNIVERSITY



Reproduced by
NATIONAL TECHNICAL
INFORMATION SERVICE
Springfield, Va. 22151

64

BIBLIOGRAPHIC DATA SHEET		1. Report No. CS 255	2. AIM 160	3. Recipient's Accession No.	
4. Title and Subtitle AUTOMATIC PROGRAMMING				5. Report Date FEBRUARY, 1972	
7. Author(s) Jerome A. Feldman				6.	
9. Performing Organization Name and Address Computer Science Department Stanford University Stanford, California 94305				8. Performing Organization Rept. No.	
12. Sponsoring Organization Name and Address National Science Foundation Advanced Research Projects Agency				10. Project/Task/Work Unit No.	
				11. Contract/Grant No. 457	
				13. Type of Report & Period Covered	
				14.	
15. Supplementary Notes					
16. Abstracts The revival of interest in Automatic Programming is considered. The research is divided into direct efforts and theoretical developments and the successes and prospects of each are described.					
17. Key Words and Document Analysis. 17a. Descriptors					
17b. Identifiers/Open-Ended Terms					
17c. COSATI Field/Group					
18. Availability Statement Release unlimited				19. Security Class (This Report) UNCLASSIFIED	
				20. Security Class (This Page) UNCLASSIFIED	
				21. No. of Pages 17	
				22. Price	

R

COMPUTER SCIENCE DEPARTMENT
REPORT CS-255

AUTOMATIC PROGRAMMING

by

Jerome A. Feldman

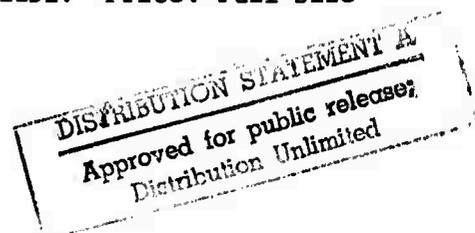
ABSTRACT: The revival of interest in Automatic Programming is considered. The research is divided into direct efforts and theoretical developments and the successes and prospects of each are described.

Note: This paper was solicited for the Communications of the ACM Anniversary issue, August 1972. Comments received before May 1 may influence the final version.

This research was supported in part by the National Science Foundation and the Advanced Research Projects Agency.

The views and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Advanced Research Projects Agency or the National Science Foundation.

Reproduced in the USA. Available from the National Technical Information Service, Springfield, Virginia 22151. Price: Full size copy \$3.00; microfiche copy \$0.95.



Automatic Programming

Introduction

The term "automatic programming" was used by some of the early designers of compilers to describe the fruits of their efforts. There is reason to believe that they were overly sanguine, but they did succeed in automating much of what programmers did at that time. The problem of parsing arithmetic expressions was a serious intellectual issue and its solution led to important theoretical and practical advances. There is currently a revival of the term "Automatic Programming" and a certain amount of work directed toward automating what programmers do at this time. This coincides with an increased amount of work on how people should write programs, discussed by Hansen in this issue.

Almost anything in computer science can be made relevant to the problem of helping to automate programming. We will suppress discussion of work on editors, file systems, numerical methods, etc., and try to point out the basic results and problems in the field. Even so, a paper of this size cannot deal adequately with the many important questions.

We begin by making a rough division of the work on automatic programming into two types. Type 1 is concerned with automating the production of programs in a particular domain of discourse. A system of this type will have considerable knowledge of the domain built in and will often be asked to produce particular answers rather than general routines. I claim that important practical advances in this area are possible with our current knowledge. Efforts of the second

type are concerned with the fundamental problems underlying the notion of automatic program synthesis. These are general and are normally restricted to generating demonstrably correct programs. Systems constructed along these lines should not be expected to be practical in the near future and are thus relegated to Section 2 of this paper.

1. Direct approaches

Even within Type 1, there are a variety of ways of viewing the problem. In its simplest form, automatic programming is just an atavistic proliferation of special purpose languages. To an extent that does not seem to be understood, special purpose languages are not only easier to use, but can be much more efficiently compiled. [13] There has been widespread use of special purpose languages in some fields [33] but subroutine packages are much more common. One reason for this is that there has not been enough additional benefit to warrant putting a special purpose language around a package of routines. If the compiler puts together the routines in an obvious way, the user might just as well do it himself. One can view Type 1 work on automatic programming as attempting to provide languages in which it will be much easier to write good programs involving large packages of routines.

A major problem one faces when trying to automate the writing of programs is this: How is one to say what is required without writing some kind of program? Workers in Artificial Intelligence have long faced this problem of process description and state descriptions. A state description for the function squareroot(z) might be:

(1) The X such that $X*X = Z$

A process description for the same function might be an ALGOL program to carry out Newton's method for the solution. Two remarks are in order.

The state description above is much simpler than any process description -- this is not always the case. It is easier to describe how to take the derivative of a polynomial than to specify a set of properties that a derivative must have. Similarly, the syntax of a programming language is given more clearly by a grammar than by a set of conditions for well-formedness. The ease of giving either a state or process description clearly depends on the language used for description.

Secondly, in writing a squareroot procedure one is forced to consider many details which are left out of (1). For example, what precision is required, are temporary cells available, etc.. Any translator which works from state descriptions will (like people) require a specification of the side conditions which constrain its choice of solutions. Notice that this virtually requires an automatic programming system to be interactive. The program will not know what values to give to side conditions and the user will not know what conditions need to be specified. We will deemphasize the question of side conditions for the remainder of this paper, but it will be an important issue in any particular design.

Now let us consider how one might design a translator for statements like (1). Many systems have a general Newton's method root finder and it would not be hard to write a compiler which recognized that (1) fit the conditions for applying this routine. One would also require a symbolic differentiation routine, but they are available. If the

function to be inverted could not be differentiated, a numerical solution could be attempted. The range of this class of compiler is impressive -- there is a vast collection of algorithms in numerical analysis and mathematical programming which could be employed. The problems of designing a syntax which allowed for the recognition of appropriate solution methods do not seem insurmountable. Rudimentary systems of this sort have been completed by Fikes [15] and Elcock et al [8].

There is a closely related line of work which has been done in the continuous simulation languages. These often provide fixed routines for solving various boundary value, optimization, etc., problems. The SLANG [36] effort is a very promising attempt to place these features in the setting of a general purpose language. There have also been a number of widely used high level procedural and non-procedural statements in general purpose languages. An example of a very sophisticated language primitive is the COBOL SORT verb. A typical statement might be:

```
SORT FACULTY ON ASCENDING RANK;  
ON DESCENDING AGE;
```

One can also specify additional keys, and procedures and files for input and output. The description of a file is written in the DATA division of COBOL and a description of the equipment available for sorting is described in the ENVIRONMENT division. The COBOL compiler selects a sorting method based on all this information. The business oriented languages also have powerful constructs for file handling and report generation [33]. None of these has been designed with all the generality and consistency one could desire, but they have proved very useful to business programmers.

We anticipate that Type 1 automatic programming research will succeed in producing systems which make programming in specific domains much easier. There are two lines of research to be pursued -- prototype domain languages and tools for building such languages.

The domain for automated programming that has received the most attention is the management information area. The goal has been to permit non-programmers to specify fairly complex calculations on large data bases. The low success/effort ratio should serve to warn us of ultimate difficulty of the task, but shouldn't prevent work on more circumscribed domains. Management information systems immediately encounter the problem of natural language communication [7] which can be avoided in many instances. There are many large groups of computer users (e.g. organic chemists, payroll programmers) who would be willing to use an artificial language if it met their needs. There are even tightly restricted domains like the Brookings model of the economy or the NCAR weather model which might justify an automatic programming effort. The idea is to combine some of the techniques discussed below with domain-specific knowledge to produce systems which will help people describe what they want the machine to do. Inevitably, such specific projects will feed back ideas to the technique-oriented research described here and the theoretical efforts discussed in Section 2.

There is a common name for a program which translates a high level description of a process into machine language -- a compiler. Compilers are among the best understood of programs and this understanding is one of the cornerstones of automatic programming research.

A modern compiler incorporates a large set of ideas for parsing and understanding programs and for producing output which efficiently carries out the computation specified [21]. The work on global code optimization [1] has been particularly important in providing ideas on how to represent and manipulate computations. There is at least one automatic programming effort [5] which is primarily concerned with optimization of high-level procedural problem statements.

The introduction of complex data structures and their operators in this generation of standard languages (PL/I, Algol 68) and the extensible language efforts are also important steps towards higher level programming. The other relevant topic from systems programming is Translator Writing Systems (TWS) [11]. The concentrated effort on TWS a few years back can be viewed as an attempt to provide tools for building special purpose languages. The problem may have been that they were not sufficiently ambitious -- the languages constructible did not have enough advantages to make their construction attractive. The addition of domain-specific knowledge and some techniques from Artificial Intelligence to the ideas developed in TWS research should provide the basis for systems support for Type 1 automatic programming efforts.

Artificial Intelligence laboratories, especially those with robot projects, have been conducting research of great relevance to automatic programming. The root problem is that a robot (even if it is only a hand-eye) will have to plan and carry out courses of action (strategies). The automatic strategy generation problem is the analog of the automatic programming problem. These are various doctrines on how to attack this problem, the most developed of which is STRIPS [16]. The existing

efforts have all been quite primitive giving rise to strategies that do not have loops and that normally do not have even conditional statements in them. The strategy and program-writing efforts will probably diverge. The strategy efforts will have to cope with incomplete information, error recovery and the other vagaries of the physical world at a much earlier stage than automatic programming ones. Perhaps more importantly, there are languages (Planner [24], POP-2, [39], QA4 [32], SAIL [14]) at the Artificial Intelligence laboratories which are continuously evolving to meet the needs of the strategy problem. Currently considered important are varied data structures, associative memory, pattern matching, automatic back-tracking, concurrent processes and the procedural representation of knowledge. I suspect that these same concepts will prove to be crucial in producing automatic programming systems. These artificial intelligence languages are also considered to be alternatives to the more abstract theorem-proving systems discussed in Section 2 for a wide range of tasks. One can view these languages as a special purpose languages for writing automatic programming systems. The following example QA4 [32] statement is illustrative. It is the recursive definition of a function SORT, which sorts a linear list

(bag):

(a) $SORT = CASES(\lambda [] , \langle \rangle) ;$

$$\lambda X \cdot B \uparrow MIN \langle X, B \rangle , X \cdot SORT(B)$$

There are two cases. The first one (up to the ";") specifies that if the argument is empty then the empty bag is the value of SORT. The general case is written in terms of the pattern matching facilities of

QA4 . The goal is to find a decomposition of the bag into an element X and a bag B such that $(\dagger) X$ is not larger than any element of B . Then the value of the function is the bag formed by prefixing X to the sorted version of B . The use of pattern matching frees the user from deciding how to find the smallest element, but normally gives rise to an inefficient algorithm. QA4 will allow one to write a faster program by specifying more details. A better (and much more difficult to achieve) solution is to have the system compile efficient programs from statements like (a) . This question of code optimization will arise yet again in Section 2.

More broadly, much of the artificial intelligence work in automatic problem solving is pertinent to the specific problem of automatic programming. This was understood by Simon [34] who made an early study of the automatic program generation problem. The article by Feigenbaum in this issue provides a good entry into the current artificial intelligence literature.

There are many other ideas which will also be useful to builders of automatic programming systems. In addition to computer science developments, the problem as here defined can exploit any systematic advance within a subject domain. One of the best examples of this kind of achievement is the MATHLAB [30] system for symbolic mathematics.

There are a number of issues which cut across the somewhat artificial distinction we have made between the systems of the first type and the systems of the second type. The most important of these is the issue of process description versus state description. A second common thread is the idea of user interaction. Neither of the systems of the

first type or of the second type seem possible today without on-line interaction. It is simply impossible for the program-writing program to know what the user wants nor for the user to anticipate all of the questions that the program-writing system may ask about his task specification. A third common theme is the idea of attaching extra information to the statement of the problem so that side conditions or predicates that must be satisfied can be added to a program. This idea was present in the frequency statements to help optimization of early compilers. More recently, Lowry [27] has suggested using range statements (e.g. this variable takes only values from 1 to 4) to aid in both error detection and optimization. Assertions in addition to or in place of procedural statements play a central role in theoretical studies of automatic programming.

II. Theoretical Studies

Much of the impetus for the renewed interest in automatic programming came from the demonstrations by Waldinger [38] and Green [20] that theorem proving programs were capable of producing simple programs. There has been a great deal of attention devoted to solving problems of the general form:

(2) Find $F(x)$ such that $R(x, F(x))$

where R is some fixed relation. For example, we could specify that we wanted a square root routine by saying

(3) Find $F(x)$ such that $F(x)*F(x) = x$.

Although statement (1) treated in most general form is equivalent to statement (3) the Type 2 approach to the problem would be quite

different. To solve the Type 2 problem, a system would have to have axioms for computer arithmetic and be able to constructively prove that there was a program which converged (presumably with an assumed accuracy) to the square root for all possible input values. This problem is much more complex than anything that has been actually attempted with a Type 2 approach. More typically the programs attempted are in a domain with simple axioms, although the logic of the program produced may be involved. A typical example is the following one from Green [20].

The problem is to construct a LISP program to sort a list. The theorem proving program must be given the properties of various LISP functions in terms of axioms. These axioms describe the effects of the functions when applied to lists. We also provide a statement of the desired result in terms of a theorem. The theorem prover then attempts to prove the theorem through a sequence of applications of the axioms. If a proof is found, the sequence of proof steps can be mapped into a sequence of function applications which constitutes the desired program.

We will consider in detail only the simpler problem of constructing a program for arranging a pair of atoms in increasing order. Green uses ten axioms for LISP, typical ones being

- (4)
- L1. $x = \text{car}(\text{cons}(x,y))$
 - L2. $y = \text{cdr}(\text{cons}(x,y))$
 - L6. $x = \text{nil} \supset \text{cond}(x,y,z) = z$
 - L7. $x \neq \text{nil} \supset \text{cond}(x,y,z) = y$

One must then state the condition which the program is to satisfy. In this simple case we define a predicate $R(x,y)$ which applies to two

pairs x,y and is true iff y is a sorted version of x (this is an instance of (2) above).

$$(5) \quad R(x,y) \stackrel{\text{df.}}{=} [[\text{car}(x) < \text{cdr}(x) \supset y = x] \wedge$$

$$[\text{car}(x) \not< \text{cdr}(x) \supset \text{car}(y) = \text{cdr}(x) \wedge \text{cdr}(y) = \text{car}(x)]] .$$

Finally, we must specify the theorem whose proof will result in the desired program. It is:

$$(6) \quad (\forall x)(\exists y)R(x,y)$$

Given the axioms in (4), the definition (5) and a definition of $<$, the program was able to prove the theorem (6) by supplying the answer:

$$(7) \quad y = \text{cond}(\text{car}(x) < \text{cdr}(x), x, \text{cons}(\text{cdr}(x), \text{car}(x)))$$

or in more familiar notation:

$$y = \text{if } \text{car}(x) < \text{cdr}(x) \text{ then } x \text{ else } \text{cons}(\text{cdr}(x), \text{car}(x)) .$$

After deriving this function for sorting a pair of numbers, Green goes on to show how a program for sorting arbitrary lists can be constructed.

For this purpose we need a predicate $RL(x,y)$ testing if y is a sorted version of x for arbitrary lists. The important step is to add an induction axiom [29] which enables the program to prove correctness for arbitrary length lists. In Greens system the user was required to specify the particular induction axiom, viz.

$$(b) \quad [RL(\text{nil}, \text{SORT}(\text{nil})) \wedge (\forall x) [\sim \text{ATOM}(x) \\ \wedge RL(\text{cdr}(x), \text{SORT}(\text{cdr}(x))) \supset RL(x, \text{SORT}(x))]] \\ \supset (\forall y) RL(y, \text{SORT}(y)) .$$

This states that if the desired function `sort` has the property that it sorts the empty list, i.e. $RL(\text{nil}, \text{SORT}(\text{nil}))$ and if RL holds for the

cdr (tail) of a list it holds for the whole list, then sort is the function which makes R1 hold for arbitrary lists. Given this axiom the program was able to come up with a sort program for lists. The problem is much more involved than we have indicated and he had to use great care in breaking the problem into pieces his program could handle.

The current state theorem proving approach to program synthesis is found in Manna and Waldinger [29]. They concentrate on a very difficult problem which is central to automatic programming -- repetition. All interesting programs have iterations or recursions, usually of dynamically determined length. The choice of which form of repetition to use and how to use it is (with the related question of data structures) among the most important parts of program synthesis (by humans or machines). Manna and Waldinger point out how certain problems give rise naturally to certain repetitive structures and how these structures are naturally represented by different induction axioms. The proper choice of induction axiom is crucial for a program of this type. Demonstration programs are constructed using the counting up and counting down version of Peano's axioms for the integers and for list axioms like (b) above. No program has yet been constructed which can choose among a large set of induction axioms, but there is work in progress on this problem.

The theorem proving approach is obviously closely related to the work on program verification which is discussed by Manna [] in this issue. If we are to have an automatic program checker, it will have to be told what the program is supposed to do. This description must itself

specify the desired result, so one might hope to have the program generated automatically. In fact, the program verification problem is easier than the synthesis problem and is much further along. Floyd [19] has suggested using the state-of-the-art in both areas in an interactive system to help people construct demonstrably correct programs. A related issue is the formal translation of programs into more efficient ones. The translation of recursion to iteration is the primary concern [35].

The general program-writing problem as stated in (2) is clearly recursively unsolvable. Even when it is solvable, the program required may be arbitrarily large [6]. There is another line of theoretical work which provides partial solutions to these difficulties, while encountering several of its own. This is based on the notion of learning (inferring) a program from examples of its behavior.

This is theoretically feasible because of an apparently paradoxical result on the inference of programs. Although it is undecidable whether a given program produces some output, a machine can find the best program which does so. The formal development is beyond the scope of this paper [10], but we will outline the basic idea. Suppose we say that the complexity of a program on an input-output pair is the product of its size and the time it takes to compute the value of the output given the input. Suppose we have all the programs enumerated by size. Then the machine proceeds as follows. Let P_1 (the first program) run for one second on the input, then let P_1, P_2 both run two seconds and so on. Eventually some program will halt with the right answer. This establishes

an upper bound, K on the complexity of the best program. Any program of size greater than K can not be the best one. For the finite number of smaller programs, the machine simply lets each one run until its space-time product (complexity) exceeds K and then chooses the best value of complexity. This algorithm, while proving the claim, is so inefficient as to defy even contemplating its implementation. There are attempts to develop reasonable algorithms for inferring programs as has been done for grammars [4]. If these work out, the inference method has several advantages.

First, the method will always yield the best program over a finite domain, and the same method can be shown to have good properties in the limit for countable domains [10]. If a direct method for solving (3) for F fails the following strategy could be applied. Use the inference method to compute a program P which works for the specific values known to obey $R(x,y)$. Given a new value x' compute $R(x', P(x'))$. If it is true then P also works for x' . If not, solve explicitly for a value y' such that $R(x', y')$ by numerical or search techniques, infer a new program P' which has $P'(x') = y'$ and continue. This entire procedure will work in many cases where theorem proving techniques would not and has at least theoretical interest. Inference techniques also have the obvious advantage that they can be used when only examples of the input-output pairs are given. Other inferential methods are being considered by Amarel [2].

The abstract work is meant to uncover basic principles which underly the problem. The people who work in this area fully realize that for practical solutions, their ideas will have to be combined with

those of the first type, incorporating specific knowledge of the domain begin treated. In fact, the system of King [25] and the proposed system of Floyd [19] are based on the use of domain-specific rules of inference and most Type 2 efforts are becoming concerned with efficient strategies for proofs in restricted domains. This brings them in close contact with the artificial intelligence languages designed to be used for searching solution spaces. The pattern matching, backup, etc. of these languages is well suited for writing directed proof procedures. The central problem is the representation of specific knowledge in a way that will be simple enough for programs to manipulate, but rich enough to efficiently direct the problem solving program.

There will never be a "solution" to the automatic programming problem. Consider the following simple statement over the positive integers:

(8) Find A, B, C, N such that $N > 2$ and $A^N + B^N = C^N$

There are, however, specific lines of work which promise to yield practical benefits or insights into the nature of programs. One can hope that this spurt of interest in automatic programming will be as fruitful as the last.

References

- [1] Allen, F., "Program Optimization," Annual Review of Automatic Programming, v. 9, 1968.
- [2] Amarel, S., "Representation and Modelling in problems of program formation," Machine Intelligence VI.
- [3] Biermann, A., "On the inference of Turing Machines from sample computations," CS241, Stanford Computer Science Department, October 1971.
- [4] Biermann, A. and J. Feldman, "A Survey of Grammatical Inference," in S. Watanabe, (Ed.), Frontiers of Pattern Recognition.
- [5] Cheatham and Wegbriet, "A Laboratory for the Study of Automating Programming," Proceedings AFIPS SJCC, 1972.
- [6] Constable, R. L., "Constructive Mathematics and Automatic Program Writers," Proceedings IFIP Congress, 1971, Ljubljana TA 137-141.
- [7] Dodd, G. G., "Elements of Data Management Systems," Computing Surveys, V1 H2, June 1969.
- [8] Elcock, E. W. et al, "ABSET: A programming language based on sets' motivation and examples," Machine Intelligence VI, Meltzer, [Ed.], Edinburgh University Press (1971).
- [9] Falkoff, A. D. and K. E. Iverson, "APL/360 Users Manual," IBM Corp., Yorktown Heights, New York, August 1968.
- [10] Feldman, J. A., "Some Decidability Results on Grammatical Inference and Complexity," Information and Control, 1972.
- [11] Feldman, J. A. and D. Gries, "Translator Writing Systems," CACM, Vol. 11, No. 2, February 1968.
- [12] Feldman, J. A. and P. D. Rovner, "An Algol-based Associative Language," CACM, Vol. 12, No. 9, August 1969.
- [13] Feldman, J., "Towards Automatic Programming," Preprints of the NATO Software Engineering Conference, Rome, Italy, 1969.
- [14] Feldman, J. and R. Sproull, "System Support for the Stanford Hand-Eye System," Proceedings Second IJCAI, London, 1971, pp. 185-190.
- [15] Fikes, R., "REF-ARF: A system for solving problems stated as procedures," Artificial Intelligence 1, (1970), pp. 27-120.

- [16] Fikes, R. E. and N. Nilsson, "STRIPS, A new approach to the application of Theorem Proving to Problem Solving," *Proceedings Second IJCAI*, London, 1971, pp. 608-621.
- [17] Floyd, R. W., "Non-deterministic Algorithms," *JACM* 14, 4 (Oct. 1965) pp. 636-644.
- [18] Floyd, R. W. "Assigning Meanings to Programs," Proceedings Symp. Appl. Math., Amer. Math. Soc., v. 19, (1967) pp. 19-32.
- [19] Floyd, R. F., "Toward Interactive Design of Correct programs," *Proc. IFIP Congress 1971, Ljubljana*, pp. 1-5.
- [20] Green, C. D., "Application of Theorem Proving to Problem Solving," *Proceedings IJCAI, Washington D.C., 1969*.
- [21] Gries, D., Compiler Construction for Digital Computers, Wiley, 1970, New York.
- [22] Gries, D., "Programming by Induction," TR 71-106, Computer Science Department, Cornell U., September 1971.
- [23] Hewitt, C., "PLANNER: A Language for Proving Theorems in Robots," *Proceedings IJCAI*, pp. 295-301, (May 1969).
- [24] Hewitt, C., "Procedural Embedding of Knowledge in Planner," *Proceedings Second IJCAI, London, 1971*, pp. 167-185.
- [25] King, J. C., "A Program Verifier," *Proceedings IFIP Congress 1971, Ljubljana TA 142-146*.
- [26] London, R. L., "Bibliography on Proving the Correctness of Computer Programs," Machine Intelligence I, B. Meltzer, (Ed.), New York, Elsevier, 1970.
- [27] Lowry, K. S., "Proposed Language extensions to aid coding and analysis of large programs," IBM TROO. 1940-1, SDD Poughkeepsie, November 21, 1969. Also NATO Conference on Software Engineering, Rome, Italy, 1969.
- [28] Manna, Z., "Properties of Programs and the First-Order Predicate Calculus," *JACM*, Vol. 16, No. 2, April 1969.
- [29] Manna, Z. and R. Waldinger, "Toward Automatic Program Synthesis," Comm. ACM, 14,3 (March 1971), pp. 151-166.
- [30] Martin, W. A. and R. Fateman, "The MACSYMA System," *Proceedings Second Symposium Symbolic and Algebraic Manipulation, ACM, New York, 1971*.

- [31] Milner, R., "Logic for computable functions, Description of a Machine Implementation," SIGPLAN Notices 7, 1(Jan. 1972), pp. 7-15.
- [32] Rulifson, J., R. Waldinger and J. Derksen, "A language for writing Problem-Solving programs," Proceedings IFIP Congress, 1971, Ljubljana, pp. 111-116.
- [33] Sammet, J. E., Programming Languages, Prentice Hall, Englewood Cliffs, New Jersey, 1969.
- [34] Simon, H., "Experiments with a Heuristic Compiler," Journal ACM, October 1963, pp. 482-506.
- [35] Strong, H. and S. A. Walker, "Properties preserved under recursion removal," SIGPLAN Notices, 7, 1 (January 1972), pp. 97-104.
- [36] Thomas, J. M., "SLANG, A Problem Solving Language for Continuous-model Simulation and Optimization," Proceedings 24 Conference ACM, 1969, pp. 23-41.
- [37] Wagner, R., "Some Techniques for Algorithm Optimization," Ph.D. Thesis, Carnegie-Mellon University, 1968.
- [38] Waldinger, R. J. and R. C. T. Lee, "PROW: A Step Toward Automatic Program Writing," Proceedings IJCAI, D. E. Walker and L. M. Norton, (Eds.), May 7-9, 1969, Washington, D.C.
- [39] Burstall, R. M. and Pollestone, R.J., Pop-2 papers, Edinburgh, Edinburgh University Press, 1968.