WORKING PAPER 67

# SOME PROJECTS IN AUTOMATIC PROGRAMMING

by

IRA GOLDSTEIN and GERALD JAY SUSSMAN

Massachusetts Institute of Technology

Artificial Intelligence Laboratory

April 1974

## Abstract

This paper proposes three research topics within the general framework of Automatic Programming. The projects are designing (1) a student programmer, (2) a robot programmer and (3) a physicist's helper. The purpose of these projects is both to explore fundamental ideas regarding the nature of programming as well as to propose practical applications of AI research. The reason for offering this discussion as a Working Paper is to suggest possible research topics which members of the laboratory may be interested in pursuing.

Work reported herein was conducted at the Artificial Intelligence Laboratory, a Massachusetts Institute of Technology research program supported in part by the Advanced Research Projects Agency of the Department of Defense and monitored by the Office of Naval Research under Contract Number N00014-70-A-0362-0005.

# SOME PROJECTS IN AUTOMATIC PROGRAMMING

## 1. INTRODUCTION

An important goal of automatic programming (AP) research is the design of computer systems capable of planning, writing and debugging programs. This paper discusses several projects which provide manageable and profitable domains in which to explore this problem.

> We shall not attempt an overview of the entire automatic programming field and, in particular, will not discuss the equally valuable projects oriented around certification, high-level language design, programming assistants and expert systems capable of natural language interactions.

The flavor of the type of research we propose is given by the following list of projects:

1) designing a system capable of writing and debugging programs for a robot manipulator.
2) designing a system capable of learning to program by being educated.
3) designing a system capable of helping a physicist or engineer to program the solution to numerical display problems.

The common thread of these projects is that they all involve problems fundamental to being a programmer. They are intended primarily to elucidate the nature of the programming process, although it should be noted that all of these domains generalize into more complex problems that have practical economic consequences. They are not "toy" problems nor are they "dead ends".

We shall explore each of these projects in the following pages. But before entering into the details of that discussion, it is worth noting five concepts which supply an underlying unity to this research--planning, debugging, annotation, learning and expertise. In the following paragraphs, a brief discussion of each of these ideas is offered.

Debugging:

Debugging is not an ability that is required only when a program has been incorrectly written: it is also a necessity in a program-writing system whose approach is based upon an interaction between proposing and testing. Such a design has the important virtue that the program writer is not faced with producing a complex procedure with total success the very first time. Program writing is simplified by being factored into a process of planning trial solutions and debugging.

> An additional dividend is that such a system is in a position to modify previously written programs in response to new demands or resources.

This paradigm is a natural one. The design of operating systems, large programs like MACSYMA and even smaller programs have this quality of proposal and repair.

There is another sense in which this paradigm of proposal and repair is essential. The user, initially, may not have an exact and complete notion of his goals. The technique, then, is to propose a tentative set of demands, generate a program and observe its performance. This will often be iterated several times as the user becomes more aware of his requirements. An AP system capable of debugging is essential for supporting this evolutionary type of development. See Minsky's discussion of "Why Programming is a Good Medium for Expressing Poorly and Sloppily Formulated Ideas" [Minsky 1966].

Other approaches to automatic programming are based upon the design of high-level programming languages and the problem of certifying that a particular program or system performs as advertised. These are valuable and offer many practical benefits. However, we believe that the problem of a computer system designing its own programs cannot be solved unless the system has the capability to debug programs that do not quite succeed. Without this ability, the system is helpless when a proposed solution fails. Yet occasional failures cannot be avoided, since the use of such essential techniques as detail-free planning, generalization and simplification inevitably result in bugs. (See [Sussman 1974] on the Virtuous Nature of Bugs.) Thus debugging skill is necessary if the system is to escape the onerous burden of always being right the first time.

Planning:

The first stage in proposing a program is to come up with a trial solution from the task description on the basis of either previous solutions to related problems or general planning paradigms.

An AP system should certainly have algorithmic descriptions of important programming techniques. Typical examples would include cognizance of optimal programs for sorting, calculating roots of polynomicals, and managing databases. These algorithms would be indexed by their purpose, performance characteristics and implementation requirements. Ideally, an AP system should have a working knowledge of Knuth.

General planning skills, however, are also necessary: otherwise the system would be helpless in tha face of a new problem that did not quite match any indexed technique. Unfortunately, it is more difficult to embody general planning skills in effective ways than it is to catalog known algorithms. Nevertheless, a start at collecting powerful problem solving ideas for converting declarative task descriptions into programs has been made [Sussman 1973, Goldstein 1974]. Three such concepts are linearity, recursion and iteration. The last two are not only control structures but also important techniques for organizing a task into manageable sub-goals.

> LINEARITY - First divide the problem into independent subgoals and design subprocedures for each sub-goal. Second, order the subgoals into some sequence and design any necessary interfaces between subgoals. (The important heuristic of linearity is that the concern for interactions can be postponed and the sub-goals can be solved independently as a first step.)
>
> RECURSION - Divide the problem into (a) a generic goal solved in terms of simpler cases of the same goal and (b) solutions for the simplest cases.
>
> ITERATION - Divide the problem into a generic goal to be accomplished by the body of a loop. Determine the number of iterations from the number of times this goal must be satisfied to complete the task.

A fundamental goal of the research described in this paper is to understand these and other planning techniques in procedurally effective ways, i.e. in ways that can support a program-writing system on tasks whose solutions have not been previously met. The STUDENT PROGRAMMER is particularly important to explore these planning paradigms since it specifically assumes that the system is not already an expert with knowledge of Knuth at its fingertips.

## Annotation:

Annotation is the generation of copious commentary describing the purpose and prerequisites of code. Such commentary serves not only to guide verification and monitoring (that the code achieves its intended purpose) but also to suggest corrections to a debugger in the event that the code fails.

Description of the effects of running a program is also necessary for both verification and debugging. The use of multi-layered databases as in CONNIVER [McDermott and Sussman, 1972] and QA4 [Rulifson 1972] provide powerful formalisms in which to model a changing environment. Systems must have "careful evaluation" modes in which a record of the execution of the program is kept. This record is used to decide whether the code succeeded and, if not, how to repair it.

All of the research projects in this paper are intended to aid in the design of a language for program commentary that supports both certification and debugging as well as in the improvement of "careful evaluators".

## Learning:

Ideally, one would like a system that learned from past problems and improved with performance. This is a very difficult goal. Yet its long-term significance is obvious. Sussman's HACKER program is the best example of such a system at present but much development will be necessary before its concepts are of practical use. Both the STUDENT PROGRAMMER and the ROBOT CONTROLLER are intended to further investigate procedural learning strategies.

## Expertise:

Domain-dependent expertise is necessary in an AP system. The research question is how best to represent such knowledge to support the construction of programs. Can the process of learning such knowledge be simplified or must an immense effort go into designing an AP system for each problem domain? What is the trade-off between domain expertise and general programming skill? The answer to these questions is not known.

The answer, however, cannot be found by choosing trivial problem domains. The ROBOT CONTROLLER and PHYSICS HELPER explore the interaction between AP and two significant, difficult problem areas.

## 2. THE ROBOT PROGRAMMER

Goal:

To build an AP system--the ROBOT PROGRAMMER--capable of tailoring control packages for robot manipulators so as to manufacture specific items, repair devices and operate machinery.

Rationale:

The outputs of such an AP system are genuine programs. These programs consist of instructions for the arm which (1) may be run many times and (2) will almost certainly have bugs either because the program is inadequate or because the environment contains some surprise.

The practical need for such a system in industrial automation is made clear by considering the primitive methods used to train UNIMATE arms. A human operator must move the UNIMATE through some sequence of actions. This direction is completely inadequate to instruct the robot to handle slightly different situations, objects or surprises.

The ROBOT CONTROLLER currently under development as part of MIT-AI's mini-robot project is a programming language for interacting with a robot arm and eye. The controller supplies high-level primitives for moving the arm, holding objects, manipulating tools and sensing the environment. For a particular task such as circuit board construction or repair, the job of the ROBOT PROGRAMMER would be to make such decisions as:

(1) which primitive should be used to achieve each sub-goal;

(2) in what order should certain sub-goals be achieved;

(3) what amount of force (input to the arm) should be used.

These are programming decisions. They represent the use of procedural knowledge involving constraints on inputs, prerequisites, ordering, and interfacing. The most interesting solution would be a system that was clearly factored into programming knowledge and arm-manipulation knowledge. Research by Goldstein [1974] and Sussman [1973] indicate that this is possible but that the effective representation of domain-dependent knowledge is crucial and cannot be ignored.

Debugging becomes an important ability when a particular control package fails because of a change in the task or in the environment. An immense reprogramming effort would hopefully not be necessary. Instead, the AP system would make the appropriate patches to the current system to take account of the new difficulties.

Previous Research:

Two recent robot problem solvers--HACKER and BUILD--are genuine automatic programmers of an elementary sort. Both of these system produce programs for a one-armed robot that direct it to build block structures. Both of these systems initially design very simple linear programs and both debug these initial plans in response to bugs.

BUILD's debugging expertise is oriented around blocks world expertise. However, the control structure of the program-writer is of general interest. Primitives "gripe" when they are unable to complete their assigned action. The gripe is passed up the stack until some higher level goal--the "gripe-catcher"--is prepared to offer a solution. The gripe-catcher is capable either (1) of attempting a different sub-goal sequence (the standard backtrack solution) or (2) of editing the current sub-goal sequence perhaps for the purpose of inserting missing prerequisites.

HACKER has less specialized blocks world expertise and more general debugging skill. The system contains knowledge of a variety of typical bugs that arise when a collection of goals is ordered into a procedure. Such knowledge consists of recognizing (1) when the goals can be reordered, (2) when it is necessary to move a prerequisite out of a particular location and to another, and (3) when there are unsolvable conflicts that require a different choice of methods. The system also contains a variety of basic design principles which include competence to do pre-planning for certain sorts of brother goals that make demands on the same resource (such as free space) and ability to subroutinize common subgoal solutions.

Milestones:

Several possible short term milestones (2-3 years) to lay the foundation for a competent ROBOT PROGRAMMER are oriented around further development of the HACKER system. These would include:

> Learning to debug programs for arches, tables, chairs for the purpose of seeing whether the concepts in the current HACKER apply to more complex programs than those required for simple towers.
>
> Building a larger catalog of bugs and patching techniques.
>
> Describing the purpose of variables and different control structures in better ways.
>
> Designing a HACKER capable of learning to be as competent as BUILD.

A longer term goal (5-6 years) would involve the interaction of the ROBOT PROGRAMMER with the ROBOT CONTROLLER for the mini-robot system. The AP task would be to tailor packages of control programs for specific industrial purposes such as the manufacture and repair of small devices. Industrial automation will not succeed if the task of programming the automata proves to be too expensive or difficult. The

ROBOT PROGRAMMER would solve this difficulty by embodying both programming and control expertise. Input to the system would be a description of the task and objects, probably by means of both English and visual descriptions. The ROBOT PROGRAMMER would be able to write programs that make provision for such typical bugs as drift and collision. It would also have knowledge of how to make effective use of the computational resources available to the ROBOT CONTROLLER such as space, time and reliability.

## 3. THE STUDENT PROGRAMMER

Goal:

To build an AP system that would become an expert programmer by being educated. The goal would be to design a system capable of gradually acquiring skill through learning the material taught in an introductory programming course.

Rationale:

A good programming course is designed to introduce the students to the fundamental concepts of computer science, provide experience through exercises and offer a foundation for tackling more difficult programming problems. Hence, there is a basis to believe that a computer who had mastered the ideas of such a course would be in a position to modify and extend itself. This represents the genetic or ontological method which posits that a means to understanding expertise is through its development.

This project is more oriented towards fundamental research than immediate practical applications. However, a difficulty of expert systems is the effort needed to modify or extend them. Hence, in the long-term, research designed to elucidate how an AP system can assume the burden of learning to solve new programming problems is of fundamental importance.

Previous Research:

A primary goal of the LOGO project at the MIT AI Laboratory has been to study how programming, planning and debugging can be best taught (and learned). Many introductory programming courses are not very successful at teaching programming. However, the LOGO environment--language, projects, concepts--has been remarkably successful in teaching programming to students of all ages ranging from elementary school to college. This success is indicated by the facility with which the students are soon able to tackle projects on their own. The STUDENT PROGRAMMER project will benefit from this by choosing as one milestone the conversion of a variety of the concepts which LOGO has identified as being fundamental to programming skill to machine-understandable form. Also, a natural choice for the training sequence for the STUDENT PROGRAMMER will be the order in which these concepts are presented by LOGO. For a deeper discussion of this, see papers by Seymour Papert [Papert 1971a, 1971b].

Three recent theses at MIT provide further background and foundational ideas for this research. These are by Sussman, Goldstein, and Ruth. Goldstein's system debugs elementary graphics programs written by beginners. Sussman's system writes and debugs programs for a one-armed robot. Ruth's program debugs sorting programs written by beginners.

Milestones:

Initially, it would be necessary to design some performance systems that were capable of planning, writing and debugging the programs typically assigned in an introductory programming course. The actual problems and competence of students in programming courses provides a real yardstick to judge the success of this research. As an example, the major projects of 6.030, the introductory course in programming for computer science majors at MIT, are:

(1) Calculate PI by rectangular approximations;

(2) Calculate the nth root of a number by Newton's method;

(3) Construct a random number generator;

(4) Write a program for computing the area of contiguous "islands" of 1's on a bit map;

(5) Write a TICTACTO program.

Eventually, it should be interesting to examine how much more the system needs in terms of general problem solving skills and specific programming competence to move on to the problems assigned in more advanced courses such as Computational Methods, Programming of Small Scale Computers and Operating Systems.

The long-term goal is for the system to become independent of classroom exercises and be capable of solving programming problems which it would actually meet as a computer programmer professional. This requires that it learn general programming paradigms from the exercises. The system must be capable of abstraction, analogous reasoning and skill acquisition. Recent research by Sussman [1973] provides insight into the skill acquisition process. The importance of the debugging approach is that novices (whether automatic or human) acquire competence through the debugging of their initially incorrect solutions. Thus, debugging skill for the student programmer will be exhibited by the fashion in which it learns to debug itself, and gradually produce more complex programs.

Better Human Programmers:

The basic goal of AP research is to make the cost of software less--both in design and maintenance. Another benefit from research into the design of a student programmer will be insight into how to better train human programmers. The design of a student programmer will involve research on the nature of learning, precise description of important computational concepts, and the creation of proper training sequences: all of which should have an impact on the education (and, ultimately, cognitive power) of human programmers both through suggestions for an improved computer science curriculum and through the creation of intelligent monitors for aiding students.

Conclusions:

Although it is true that people have important generalized learning abilities--and we wish to understand them as part of this research--it is also true that the programmer knows a great deal about computers and algorithms. This would include a bag of canned solutions, debugging skills, abstract concepts about the form and purpose of programs, data structures, and control. Most programmers do not invent recursion, they are told about it. This research is directed towards making these programming ideas explicit and machine-understandable.

The reader may feel critical of this project because it focusses on the beginner rather than the expert. Indeed, it is true that writing an AP system that captures the capabilities of the expert programmer would be desirable. However, experts are not always articulate about their abilities. By observing the development of programming skill, it may be easier to see which knowledge is fundamental and which irrelevant.

However, the most important reason for this project is to build a model of the learning process. The human expert is even less helpful at describing his learning mechanisms than at describing his field of expertise. The basic goal of AP research is to free the user from writing particular programs. The most exciting way that this goal might be met is with a system that had an ability to learn in new situations. Such a system would not only free the user of writing programs, it would also free the designer of the AP system from constantly having to improve it. This is very ambitious but, we feel, worth undertaking as part of the theoretical side of the current AP effort.

## 4. THE PHYSICS HELPER

Goal:

To build a program capable of helping with the everyday programming chores of a physicist or engineer. The program would have to plan, write, and debug programs for the computation and display of data useful to such an individual. The PHYSICS HELPER would be entrusted with the maintenance of a library of useful procedures indexed by problem types for which they are appropriate along with techniques for retrieval of these routines, piecing them together for particular problems, and preparing useful data-bases.

Rationale:

The MACSYMA system is a model of expertise; however, it suffers from a lack of internal self-description. The user cannot ask it questions regarding what methods it recommends for a given problem. Instead, he must essentially be both an expert in his particular domain as well as in programming. The solution is for MACSYMA to be capable of generating and comprehending comments; knowing general planning strategies; maintaining a library of solutions indexed by purpose; and debugging and generalizing these routines on the basis of experience. The PHYSICS HELPER will contribute to an understanding of these problems by confronting them in a non-trivial situation.

Scenario:

A physicist who often needs plots of the equipotential lines of an electric field might have a great deal of use for such a system. He would, for example, want to see the equipotentials of a field caused by a rather complex charge distribution with some equally complex boundary conditions. In general, this problem is theoretically trivial but computationally a disaster. All one needs to do (theoretically) is:

1: Find a particular solution of Poisson's equation for the given charge distribution, without regard for the boundary conditions to be imposed. This can be done by evaluating Poisson's integral at every point for which a potential value is desired.

2: If this particular solution matches the required boundary conditions, we are done. If not, we form a new set of boundary conditions by subtracting the values of the particular solution on the boundary from the given boundary conditions, and then:

3: Solve the homogenous problem -- Laplace's equation with the new boundary conditions calculated in 2. This can be calculated by relaxation techniques, for example.

4: Return the sum of the homogenous and particular solutions.

Of course, this is really computationally quite ridiculous. One would expect such a method to take nearly forever to produce the solution of a very simple problem. A programmer, given the problem of producing the equipotentials of a group of point charges, a dipole, for example, would probably pull out of his library the potential function (or perhaps a table of computed answers) of a point charge, and compute the answer by superposition of point charge potentials, appropriately scaled and translated. If there is, for example, a boundary condition of a simple sort, such as a conducting sphere or a conducting plane, one can often simplify a problem drastically by the use of the method of "image charges". Thus, it is often unnecessary to do an unfeasible relaxation to satisfy the boundary conditions with a given charge distribution.

In order to be able to decide what really _must_ be computed, the physics helper must know lots of physics. Once the basic problem is understood, and the quantities that must be computed are known, the next question is how to efficiently compute them. The physics helper must know quite a bit about numerical analysis. It must know how to effectively compute an integral. It must know what kinds of algorithms are good for particular kinds of integrands. It must know how finely to divide the integration range to achieve the required degree of accuracy. The numerical analyst part of the physics helper must compile algorithms given specifications of what must be computed. It must know how to do relaxation or sparse matrix methods for Laplace's equation with boundary conditions. It must know how finely it must divide space to get a stable solution of the correct degree of accuracy. It must know about interpolation techniques.

Finally, the physics helper must know about the details of the implementation of algorithms in the light of the available computational resources. It must know about how to choose effective data structures for representing the computational objects referred to by the numerical analyst. It must know when it is cost effective to use marginal arrays to implement multi-dimensional indexing, and when to use direct index computations. It must know how to decide where it is appropriate to use fixed, floating point, or double-precision arithmetic; where to use arrays and where to use lists.

Subgoals:

We believe that any project as deep and complex as this one must be attacked in a multi-pronged approach each prong of which is likely to yield valuable results as well as parts of an ultimate system. Some subgoals we see as relevent are:

1. The design of a problem solver with enough knowledge of field theory to be able to specify, in terms usable by a numerical analyst, just what needs to be calculated for a particular problem -- just what special properties of the physical situation limit the computational hair.

2. The design of a problem solver who can take specifications produced by the physicist program (1) and produce numerical algorithms suitable for implementation. Again, we have a case where special knowledge of the problem domain (numerical analysis) is essential to produce feasible algorithms.

> For both sub-goals 1 and 2, we expect to utilize MaCSYMA's mathematical expertise. This will lead to addressing the issues mentioned earlier regarding the addition of problem-solving skill to mathlab.

3. The design of a specialist in numerical techniques for compilation of the algorithms produced by the numerical analyst program (2). this specialist should be capable of correctly allocating resources and making declarations to a compiler (say the MACLISP fast number compiler) which ensure the generation of near-optimal code.

4. The design of a display expert who can compile display routines appropriate to the presentation of the results required by the user.

Conclusion:

The problem domain must be suitably limited or else the tasks of designing the physicist and numerical analyst modules become, in themselves, enormously difficult. Also, it may prove that a system with properly documented and sufficiently powerful numerical routines does not require an AP system at all. However, from a more positive standpoint, we feel that MACSYMA is now at the point that it would clearly benefit from an AP module that would aid the user in constructing his program. Hence, it is worth pursuing this research to understand more precisely where the division of labor lies between domain-dependent problem solver and automatic programmer.

## 5. Bibliography

[Goldstein 1974]
Goldstein, I.P.
Understanding Simple Picture Programs
AI TR-294 MIT-AI Laboratory (April 1974)

[Martin 1972]
Martin, W.A.
A Management Systems Language
Project Mac Automatic Programming Group
Internal Memo 2 (April 1972)
See also other memos by the Project Mac Automatic Programming Group

[McDermott and Sussman 1972]
McDermott, D.V. and Sussman, G.J.
The CONNIVER Reference Manual
AI Memo 259 MIT-AI Laboratory (May 1972) (Revised July 1973)

[Minsky 1966]
Minsky, M. L.
"Why Programming is a Good Medium for Expressing Poorly Understood and Sloppily Formulated Ideas"
Design and Planning (1966)

[Papert 1971a]
Papert, Seymour A.
"Twenty Things to Do with a Computer"
AI Memo 248, MIT-AI Laboratory (June 1971)

[Papert 1971b]
Papert, Seymour A.
"A Computer Laboratory for Elementary Schools"
AI Memo 246, MIT-AI Laboratory (October 1971)

[Rulifson 1971]
Rulifson, J.F.
QA4 Programming Concepts
Stanford AI Technical Note 60 (August 1971)

[Sussman 1973]
Sussman, G.J.
A Computational Model of Skill Acquisition
AI TR-297 MIT AI Laboratory (August 1973)

[Sussman 1974]
Sussman, G.J.
On the Virtuous Nature of Bugs
Submitted to the Sussex Conference on Artificial Intelligence
(July 1974)