(12)

Stanford Artificial Intelligence Laboratory
Memo AIM-321

Computer Science Department
Report No. STAN-CS-78-696

September 1978

Recent Research in Artificial Intelligence and Foundations of Programming

by

John McCarthy, Tom Binford, Cordell Green,
David Luckham, Zohar Manna,
edited by Les Earnest

Research sponsored by

Advanced Research Projects Agency

COMPUTER SCIENCE DEPARTMENT
Stanford University

UNCLASSFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: STAN-CS-78-695, AIM-321 (14)	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE *(and Subtitle)*: Recent Research in Artificial Intelligence and Foundations of Programming (6)	5. TYPE OF REPORT & PERIOD COVERED: (9) Progress report,
6. PERFORMING ORG. REPORT NUMBER: AIM-321	
7. AUTHOR(s): (10) John McCarthy, Tom Binford, Cordell Green, David Luckham, Zohar Manna, edited by Lester Earnest	8. CONTRACT OR GRANT NUMBER(s): (15) MDA903-76-C-0206,
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Artificial Intelligence Laboratory, Stanford University, Stanford, CA 94305	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: ARPA Order 2494
11. CONTROLLING OFFICE NAME AND ADDRESS: Eugene Stubbs, ARPA/PM, 1400 Wilson Blvd., Arlington, VA 22209 (11)	12. REPORT DATE: September 1978
	13. NUMBER OF PAGES: 92
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*: Philip Surra, ONR Representative, Durand Aeronautics Guilding, Rm 165, Stanford University, Stanford, California 94305 (12) 100 p.	15. SECURITY CLASS. *(of this report)*: 15
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Releasable without limitation on dissemination

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

This report summarizes recent research in the following areas: artificial intelligence and formal reasoning, mathematical theory of computation and program synthesis, program verification, image understanding, and knowledge based programming.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE S/N 0102-014-6601 |

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

Stanford Artificial Intelligence Laboratory
Memo AIM-321

September 1978

Computer Science Department
Report No. STAN-CS-78-696

Recent Research in Artificial Intelligence and Foundations of Programming

by

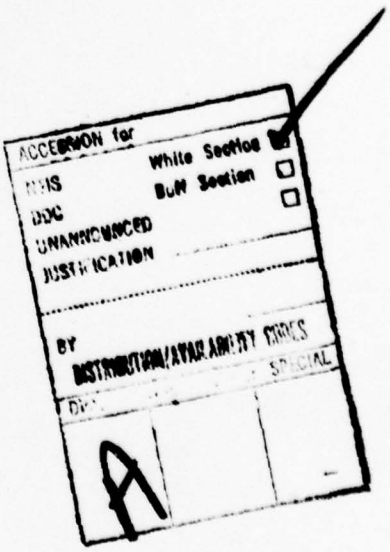
John McCarthy, Tom Binford, Cordell Green,
David Luckham, Zohar Manna,
edited by Les Earnest

Abstract

Summarizes recent research in the following areas:
artificial intelligence and formal reasoning,
mathematical theory of computation and program synthesis,
program verification,
image understanding,
knowledge based programming.

This research was supported by the Advanced Research Projects Agency of the Department of Defense under ARPA Order No. 2494, Contract MDA903-76-C-0206. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of Stanford University, or any agency of the U. S. Government.

TABLE OF CONTENTS



1. Introduction

This report summarizes five related research projects that have been sponsored by the Defense Advanced Research Projects Agency.

- *Basic research in artificial intelligence and formal reasoning* addresses fundamental problems in the representation of knowledge and reasoning processes applied to this knowledge. Solution of these problems will make possible the development of analytical applications of computers with large and complex data bases, where current systems can handle only a very restricted set of data structures and queries.

- *Mathematical theory of computation and program synthesis* studies the properties of computer programs. The goal is to provide a sound theoretical basis for proving correctness or equivalence of designs and to automaticlly synthsize programs having certain properties.

- *Program verification* is a closely related project whose goal is to improve the reliability of important classes of programs such as compilers, operating systems and realtime control systems, and to standardize techniques for program construction, documentation and maintenance.

- *Image understanding* is aimed at mechanizing visual perception of three-dimensional objects either from photographs or from passive imaging sensors. Advances in this field are expected to lead to much more efficient photointerpretation capabilities as well as automatic visual guidance systems.

- *Knowledge based programming* is an interactive approach to programming in which the computer assists the user in formulating the specifications of his problem and in designing the data structures and procedures needed to solve it.

Readers who wish to dig deeper should see the references at the end of each section. Appendices list dissertations, films, and other recent reports as well as external publications by the staff.

2. Basic Research in Artificial Intelligence and Formal Reasoning

Personnel: John McCarthy,
Richard Weyhrauch, *Student Research Assistants*: Juan Bulnes, Robert Filman, Andrew Robinson, Carolyn Talcott, David Wilkins.

Applied research requires basic research to replenish the stock of of ideas on which its progress depends. The long range goals of work in basic AI and formal reasoning are to make computers carry out the reasoning required to solve problems. These problems may be intellectual, e.g. doing mathematics, playing games or solving puzzles, or they may be of the practical sort necessary to carry out everyday events like getting to the grocery store on time. Recent discoveries have made it clearer how to apply our main tool, first order logic, both to AI and to reasoning about programs. This brings application nearer, especially to proving programs and hardware correct, and has changed the direction of some of our research.

This report begins with a brief discussion of the general problem of representing knowledge. We discuss a variety of problems in this class that need to be solved in order to make progress towards the above mentioned long range goals. We then discuss the particular areas of interest to our group and our recent accomplishments in these areas. Finally we review the FOL reasoning system and describe recent developments and applications of this system. FOL is one of our basic research tools and has been used in much of the work described below.

2.1 Fundamental problems of AI and formal reasoning

Imagine that you want to implement a system that is expected to do some reasoning. One basic problem that must be solved is that of knowledge representation. Many existing systems represent the particular facts that they deal with by entries in a data base. This, however, in not the only kind of knowledge that is needed to effectively access this data. For example, few existing systems represent general facts by entries in the data base. Instead they represent general facts by programs or by *semi-programs* like productions. This works very well for applying the general facts to particular cases, but it doesn't work well if in order to effectively use the data base you need to figure out some consequence of these general facts. In this case you need two things: first, an explicit representation of the general facts themselves; and second, the ability to deduce new general facts from old ones.

What kind of facilities do you need to represent these general facts? One important tool in our research is first order logic. Current fashion in AI research questions the rigidity of this approach. One reason for being interested in first order logic is that dealing with first order sentences is in some sense a minimal requirement for any reasoning system. Consider the list of properties a reasoning system must have:

1) it must have the ability to name objects: John and Richard.

2) it must be able to specify the parts and properties of the objects that it can name: the color of the hair of Richard.

3) it must have ways of building up a complicated object from its parts: a poker hand is built out of five playing cards.

4) it must be able to speak about the relation between things: John is taller than Richard.

5) it must have ways of building up complicated relations between objects in terms of simpler ones: (x is the uncle of y) means (y is the brother of z) and (z is a parent of x).

Any system that can express ideas like these contains as a subpart what is called by

logicians "quantifier free first order logic". If in addition you believe that it is reasonable to be able to say sentences like "all the chairs in this room are red", and "there is a pirate in the cave" then you must have full first order predicate logic. Thus, in order to represent general assertions such as those mentioned above one needs quantifiers and the most developed logical system with quantifiers is first order logic. Within first order logic, there are many possible ways of representing a particular kind of fact, and much further study is required.

Many applications of "intelligent" programs will require that decisions be made based on information obtained in a variety of ways. For data bases to include the many types of information that decision makers really need will require additional major advances in representation theory. As mentioned above current data base technology at best allows simple relations to be represented - e.g. "Smith is the supervisor of Jones." Additions from current AI techniques would allow simple generalizations of relations ("Every employee has a supervisor except the director."), but this leaves a tremendous range of representation problems untreated:

1. Mental states - what a person or group believes, knows, wants, fears, etc.
2. Modalities - what may happen, what must happen, what ought to be done, what can be done, etc.
3. Counterfactual conditionals - if something were true what else would be the case.
4. Causality - how does one event follow because of another. The preconditions of events and the consequences of events. Concurrent events and their laws of interaction and non-interaction.
5. Actions and their modifiers, e.g. "slowly". Ability - conditions under which a person or group can do something.

Facts of these kinds cannot be adequately represented in data bases at present, and there are undoubtedly other phenomena essential for intelligence which have yet to be discovered. Before such facts can be incorporated in data bases and question-answering programs in a general way, basic research must determine the logical structure of these concepts.

Our object in raising these problems is not to show that present database efforts are misdirected. In our opinion, the problems being explored are entirely appropriate. However, it is necessary to look further ahead and provide the basic research foundation for more advanced database work. The same basic research will also support other intelligent system and program verification advances, but we haven't time or space to elaborate these here.

2.2 Recent research interests and accomplishments

We now discuss the areas of particular research interest of members of the Formal Reasoning group and give some details of recent results and accomplishments. At a detailed level there is a diversity of problems studied. The problem of knowledge representation is a common underlying theme.

Knowledge and belief

The solution of real world problems frequently requires the ability to reason about other peoples knowledge and beliefs. The problems of reasoning about knowledge are more difficult than reasoning about things.

Suppose we have the sentences *Pat knows Mike's telephone number* and *Mike's telephone number is the same as Mary's*. A computerized deduction system that uses the rule that equals may be substituted for equals might conclude *Pat knows Mary's telephone number*. This is not a legitimate deduction, even though it would be legitimate to deduce that Pat dialed Mary's telephone number from the fact that he dialed Mike's number and the fact that the numbers are the same.

The fact that substitution of equals for equals is legitimate in some contexts and not in others has been well known for a very long time. Correct logical laws for handling such cases have been proposed, but the presently known solutions do not seem adequate.

● McCarthy has continued his study of the formalization of facts about knowledge. There have been two important results recently. First he has shown how to express the assumption that a person knows nothing more about a subject than knowledge explicitly ascribed to him. This permits deducing that he doesn't know something. Second he has shown how to express the effect of learning a fact on a person's state of knowledge. Third, he has shown how to express joint knowledge of several people, and finally he has shown how to axiomatize "knowing what" rather than merely the "knowing that" treated by Hintikka and other philosophers.

Conjectures

It has long been recognized that standard logic does not represent the many kinds of reasoning that people use in forming conjectures. It now appears that much human reasoning involves conjecturing that the known facts about a phenomenon are all the relevant facts.

Strict logical deduction does not permit drawing a conclusion from certain facts that would be changed if additional facts, supplementing but not contradicting them, were discovered. In logic, if a conclusion follows, it will still follow when more facts are added. Humans, on the other hand, are always drawing this kind of conclusion. We now think that machines must also reason this way, and that programs confined to strict logical reasoning must either be unable to draw conclusions or they must use axioms so unqualified that they are false.

● McCarthy's *minimization schema* work has developed into a study of conjectural reasoning, the first results of which were included in [McCarthy 1977b]. The method, now called *circumscription*, seems to be present in informal human reasoning and may be the most important logical difference between informal human reasoning and the formal reasoning of mathematical logic. [McCarthy 1978d] gives an axiom schema of first order logic called the *circumscription induction schema* which can be used to represent in a flexible way the conjecture that the entities that can be shown to exist on the basis of the information in a certain data base are all the relevant entities that exist. The flexibility comes from the fact that the set of information conjectured to be all the relevant information is readily changed.

Circumscription is a fully formal mode of reasoning and can be programmed for a computer. On the other hand, it is not *valid*, i.e. it sometimes leads to wrong conclusions. This is to be expected, because mathematicians have proved the completeness of the rules of inference of first order logic; admitting any general laws that generate conclusions not attainable by the old laws makes the system inconsistent. Therefore, programs that use circumscription cannot be certain that their results are correct and must be made capable of withdrawing applications of circumscription that lead to wrong results. This will make them more like humans - getting increased power at the price of fallibility. Its further development is essential for progress in AI.

Reasoning with observation

Many human reasoning processes involve interspersing observations of the external world with the use of logical inference. We believe that intelligent machines must also do this.

● Bob Filman has completed his thesis research [Filman 1978] where he demonstrates that the chain of reasoning involved in a

complex chess problem requires programs that observe a chess board as well as perform deductions if the solution is to be considered feasible. The point of this research was not to solve chess problems, but to explore how the ability to make direct observations of the world, in this case a chessboard, can be interspersed with deduction to better solve problems. This work was facilitated by what we have called the semantic attachment feature of FOL. His experience with observational reasoning shows that we still have only begun to understand it.

Reasoning about programs

Researchers in mathematical theory of computation have developed a number of techniques for analyzing and proving properties of programs. These techniques have proved useful but are typically capable of treating a limited class of problems. We feel that it is important to develop techniques that are capable of handling a wide variety of questions about programs in a uniform manner. The following is a collection of examples of properties that we would like to express and questions that we would like to be able to answer:

1. Parsing - is p a well formed program, is s an acceptable specification?
2. Correctness - does a program, p, satisfy some specification, s?
3. Equivalence - do two programs meet the same specifications?
4. Classes of programs - can we express the fact that a program contains some particular construct such as "while loop" or "go to"?
5. Properties of such classes - can we state properties of programs of some such class, for example that the equivalence of two programs of a particular class is decidable?
6. Lemmas - can such facts be applied to particular programs?
7. Resources - how much storage does a particular program require?

- McCarthy found Cartwright's thesis [Cartwright 1977] the key to implementing his long term goal of expressing recursive program definitions as sentences of logic. While McCarthy's older approach required using a logic of partial functions and predicates which introduces many complications, Cartwright's approach uses ordinary first order logic. Using it permitted McCarthy to simplify his earlier ideas and apply them to moderately complicated programs. Besides an exposition of Cartwright's formal ideas separated from his proof-checker, [McCarthy 1978b] contains a minimization scheme that characterizes a recursive program as the minimal solution of the Cartwright functional equation, a characterization of the verification methods of inductive assertions (Floyd) and subgoal induction (Manna-Pnueli and Wegbreit-Morris) as axiom schemata, and a greatly simplified analog of the Goedel-Kleene method of representing recursive functions in first order logic.

- Richard Weyhrauch has used reflection to implement the McCarthy minimization schema [McCarthy 1978b] for proving the correctness of LISP programs. This is a step toward our goal of constructing a system for reasoning about the correctness of LISP programs.

- A program solving the samefringe problem is given in [McCarthy 1977]. Proving the correctness of this program involves a non-trivial induction step. It was formalized using two different first order axiomatizations of LISP, (once by Richard Weyhrauch and once by Carolyn Talcott) based on theoretical ideas of McCarthy [1978b]. The proofs were constructed and checked using FOL.

We believe that these recent results are the basis for developing a system for reasoning about programs that will have the properties mentioned above. And we are beginning work on such a system using the FOL reasoning system.

Patterns

Many of the patterns that an "intelligent" program will have to recognize do not fall into the categories so far treated in AI work. For example, explaining an unknown activity of an adversary requires conjecturing a goal and its relation to other goals, a belief structure that makes the goal seem desirable and attainable, and a means of attaining the goal that gives rise to the observations. Present AI pattern recognition programs find patterns in observed data rather than introduce new entities in order to explain the data. McCarthy is developing a general notion of pattern, and Wilkins is using chess to develop some advanced notions of strategic pattern.

- Dave Wilkins is studying the problem of applying knowledge in a problem solving system. He is developing the system PARADISE (PAttern Recognition Applied to DIrecting SEarch) which finds the best move in tactically sharp middle game positions from the games of chess masters. His system employs a knowledge-based approach where much chess knowledge is stored as patterns and used as productions. He has developed a production language and methods of forming plans to guide a search.

2.3 The FOL reasoning system

The study of representation of facts and modes of reasoning has an experimental as well as a theoretical aspect. The long run test of the usefulness of a means of representation or a mode of reasoning is its contribution to the success of question answering and problem-solving programs. However, building such systems is a slow way of testing newly developed reasoning concepts, because it often requires the creation of a whole new data base format. New concepts can be studied much more quickly if we can test their consequences directly in relative isolation.

The FOL reasoning system [Weyhrauch 1977] provides an environment in which new ideas can be formulated, tested and developed. In order to check that some line of reasoning is valid, we need a formal and mechanizable notion of proof. A program that can decide whether or not a proof is valid is called a *proof checker* and forms the basis of a system for testing theories and modes of reasoning. The facts we are studying are general facts about situations, events, actions and goals, the effects of actions that manipulate physical objects, and the facts about sources of information such as books, computer files, people and observation that are necessary in order for a program to obtain the information required to solve problems. In addition to its applications to AI, we are using FOL to develop techniques to verify that computer programs meet their specifications and to study other properties of programs.

FOL can be thought of as a *conversational reasoning system*. Together, FOL and a user establish the language they will use, decide on the objects they are discussing, and agree about their basic properties, i. e. axioms. Then they discuss interactively the reasons why certain facts about these things can be concluded from others. There latter can be viewed as a form of proof checking or theorem proving.

The proof checking aspect of FOL is based on a natural deduction formulation of first order logic as described in [Prawitz 1965]. Formal proofs carried out in a pure natural deduction system are generally extremely long, because the usual logical systems do not incorporate as primitives all the modes of reasoning actually used. FOL contains many features that allow more natural and efficient expression of facts and reasoning. One of the most important is semantic attachment. This feature of FOL allows the user to create a LISP model of the object that he is wants to reason about. This allows him to conclude some facts simply by examining them, rather than doing some complicated reasoning about them. In everyday life even the weather man looks out the window to see if it is raining. He does not

conclude it is raining by some complicated reasoning based on his knowledge of weather. We call the data structure representing the combination of formal reasoning ability and observation an L/S pair (or language/simulation structure pair). In this environment proofs can be carried out by the usual methods of deduction, by computation in the model, or by combinations of both.

Frequently the objects we reason about are themselves theories, i.e. L/S pairs. This reasoning about theories is called metatheory. In fact we do meta reasoning all the time. For example, when we think about what questions we should ask a data base, we are reasoning about our theory of what is stored in the data base, not about the facts that are stored there. A formalization of the reasoning about theories provides (1) a formal description of FOL and (2) a theory in which to carry out proofs of statements about theories. An initial axiomatization, META, of the metatheory of FOL has been made and has been used in several projects.

META is just an ordinary first-order theory, and as such is represented as a data structure in FOL. This provides FOL with a certain amount of ability to reason about itself. The ability of programs to manipulate pointers and the ability of FOL to view the structure representing an L/S pair as a part of a simulation structure makes FOL in some sense self-reflexive. This kind of self-reflexive system is completely new. (Further elaboration is given in [Weyhrauch 1978b]).

FOL software

Summer 1978 closes a period of intense FOL software development. The coding was done under the direction of Richard Weyhrauch who was helped by Andrew Robinson, Chris Goad, Carolyn Talcott, Juan Bulnes and Dan Blom. The extensions and improvements described allow facts and proofs to be expressed more naturally, and make shorter proofs possible by providing simplification mechanisms, and the ability to replace some parts of a proof by computation.

- The semantic attachment mechanism, including the use of representations, was completed.

- The rewriting system was finished. This is a syntactic simplifier capable of using a user-specified collection of equalities and logical equivalences to simplify a term or formula.

- The semantic and syntactic simplification mechanisms were combined to produce a general first order expression evaluator. This turns out to be a very powerful theorem proving tool, particularly in the applications involving the use of metatheory.

- The semantic attachment mechanism was augmented by the implementation of the LET command, providing the ability to attach to an individual constant the result of evaluating a term.

- The many-sorted aspect of FOL was improved to allow the declaration of polymorphic functions.

- Data structures were developed to allow the presence of several L/S pairs and for multiple proofs within an L/S pair environment. Commands were implemented for switching attention to a particular proof or L/S pair (context switching).

- Reflection principles can be used to express the connection between statements in a theory and its metatheory Several reflection principles were implemented in the form of a REFLECT command.

Applications of FOL to problems of reasoning and representation.

More details about the FOL system and its applications can be found in [Weyhrauch 1978a,b]

- A primitive goal structure mechanism has been implemented in FOL by Juan Bulnes. He has carried out various experiments using the new goal structure. In particular a version of Ramsey's theorem was proved in about one fifth the number of steps previously required. This is a substantial improvement, although it is still somewhat longer than an informal proof would be. (Ramsey's theorem can be described as follows: given an infinite set of points such that for every pair there is either a black line or a red line connecting them; there is either an infinite subset of those points such that every pair is connected by a black line, or there is an infinite subset such that every pair is connected by a red line.)

- Weyhrauch and Talcott axiomatized D. Michie's blind robot problem as an example of reasoning about actions and moving. This work incorporates a new idea of the notion of situation. The working system has the ability to answer most reasonable questions about the robot situations in a single step.

- We have begun using the FOL formalism to represent reasoning about asynchronous actions and time and to study the question of how we get from sensory (or sensor) data to an understanding of the world. The semantic attachment mechanism of FOL plays a major role in this work.

- Todd Wagner studied the problem of hardware verification. He developed a language for specifying the behavior of digital circuits and their components. Several circuits were described and the verification of their correct behavior was carried out using FOL [Wagner 1977].

2.4 References

[Aiello 1977] L. Aiello, M. Aiello, Richard Weyhrauch, **Pascal in LCF: Semantics and Examples of Proofs**, Theoretical Computer Science 5 (1977).

[Filman 1978] Robert Filman, **The interaction of Observation and Inference**, PhD Thesis, forthcoming.

[Kelley 1955] John Kelley, *General Topology*, D. van Nostrand Company, Inc., 1955.

[McCarthy 1959] John McCarthy, **Programs with Common Sense**, *Proc. Int. Conf. on Mechanisation of Thought Processes*, Teddington, England, National Physical Laboratory, 1959.

[McCarthy 1963a] John McCarthy, **A Basis for a Mathematical Theory of Computation**, in Braffort, P. and Herschberg, D. (eds.), *Computer Programming and Formal Systems*, North-Holland, Amsterdam, 1963.

[McCarthy 1963b] John McCarthy, **Towards a Mathematical Science of Computation**, in Popplewell, C.M. (ed.), *Information processing: Proceedings of IFIP Congress* 62, North Holland, Amsterdam, 1963.

[McCarthy 1964] John McCarthy, **A Formal Description of a Subset of ALGOL**, in Steel, T.B., Jr. (ed.), *Formal Language Description Languages for Computer Programming*, North Holland, Amsterdam, 1966.

[McCarthy 1965] John McCarthy, **A Proof-Checker for the Predicate Calculus**, Stanford AI Memo AIM-27, March 1965.

[McCarthy and Hayes 1969] John McCarthy and Patrick Hayes, **Some Philosophical Problems from the Standpoint of Artificial Intelligence**, Stanford AI Memo AIM-73, November 1968; also in D. Michie (ed.), *Machine Intelligence*, American Elsevier, New York, 1969.

[McCarthy and Painter 1967] John McCarthy and James Painter, **Correctness of a Compiler for Arithmetic Expressions**, in Schwartz, J.T. (ed.), *Proc. of a Symposium in Applied Mathematics, Vol.* 19 –

Mathematical Aspects of Computer Science, American Mathematical Society, Providence, Rhode Island, 1967.

[McCarthy 1977a] John McCarthy, Another SAMEFRINGE, SIGART Newsletter No. 61, February 1977, p4.

[McCarthy 1977b] John McCarthy, **Epistemological Problems of Artificial Intelligence**, Proceedings of the 5th International Joint Conference on Artificial Intelligence, 1977(Invited Paper).

[McCarthy 1978a] John McCarthy, **History of LISP**, Proceedings of the ACM conference on History of Programming Languages, 1978.

[McCarthy 1978b] John McCarthy, **Representation of Recursive Programs in *First Order Logic***, Proceedings the International Conference on Mathematical Studies of Information Processing, Kyoto Japan, 1978.

[McCarthy 1978c] John McCarthy, **Ascribing Mental Qualities to Machines**, to be published in *Philosophical Perspectives in Artificial Intelligence*, Martin Ringle (ed.), Humanities Press, 1978.

[McCarthy 1978d] John McCarthy, **Circumscription Induction - A way of jumping to conclusions**, (forthcoming).

[McCarthy 1978e] John McCarthy, **First Order Theories of Individual Concepts and Propositions**, Stanford AI Memo, (forthcoming).

[McCarthy 1978f] John McCarthy, M. Sato, T. Hayashi, and S. Igarishi, **On the Model Theory of Knowledge**, Stanford AI Memo (forthcoming).

[Prawitz 1965] Dag Prawitz, *Natural Deduction*, Almqvist & Wiksell, Stockholm, 1965.

[Stanford 1969], Stanford University, **Proposal for Continuation of the Stanford Artificial Intelligence Project and the Heuristic Dendral Project**, proposal to ARPA, June 1969.

[Wagner 1977] Todd Jeffry Wagner, **Hardware Verification**, Stanford AI Memo Stanford AI Memo AIM-304, September 1977.

[Weyhrauch 1977] Richard Weyhrauch, **A users manual for FOL**, Stanford AI Memo Stanford AI Memo AIM-235.1, July 1977.

[Weyhrauch 1978a] Richard Weyhrauch, **Lecture Notes on Logic and AI**, Prepared for Summer School on Foundations of Artificial Intelligence and Computer Science, Pisa Italy, June 1978 .

[Weyhrauch 1978b] Richard Weyhrauch, **Prolegomena to a theory of mechanized formal reasoning**, forthcoming.

[Weyhrauch 1978c] **Proofs using FOL**, forthcoming.

3. Mathematical Theory of Computation and Program Synthesis

Personnel: Zohar Manna, *Student Research Assistants*: Nachum Dershowitz, Martin Brooks, Chris Goad.

3.1 New Verification Techniques

Nachum Dershowitz & Zohar Manna

The goal of this research is to find more powerful verification techniques that will help make program verification a more practical tool for programmers and more readily amenable to automation.

In the course of recent ARPA supported research, we have found two techniques of widespread interest that will undoubtably have an impact on future work in program verification. They are the use of intermittent-assertions to prove the total correctness of programs, and the use of multiset orderings to prove the termination of programs. We have also begun investigating the verification of production systems.

3.1.1 Intermittent Assertion Method

Manna and Waldinger [1978] explored a technique for proving the correctness and termination of programs simultaneously. This approach, which they call the *intermittent-assertion method*, involves affixing comments to points in the program but with the intention that only *sometime* will control pass through the point and satisfy the attached assertion. Consequently, control may pass through a point many times without satisfying the assertion, but control must pass through the point at least once with the assertion satisfied; therefore they term these comments *intermittent assertions*. If one proves the output specification as an intermittent assertion at the program's exit, then he has simultaneously shown that the program must halt and satisfy the specification. This establishes the program's total correctness in a single proof, while the conventional approach requires two separate proofs to establish partial correctness and termination. This intermittent assertion method, introduced by Burstall [1974], promises to provide a valuable complement to the more conventional methods.

Manna and Waldinger use the phrase

sometime Q at L

to denote that Q is an intermittent assertion at label L (i.e. that sometime control will pass through L with assertion Q satisfied). If the entrance of a program is labelled *start* and its exit is labelled *finish* , one can express its total correctness with respect to an input specification P and an output specification R by

Theorem: if sometime P at start
then sometime R at finish .

This theorem entails the termination as well as the partial correctness of the program, because it implies that control must eventually reach the programs exit, and satisfy the desired output specification.

Generally, to prove the total correctness of a program, one must affix intermittent assertions to some of the program's internal points, and supply lemmas to relate these assertions. Typically, one will need a lemma for each of the program's loops, to describe the intended behavior of that loop. The proofs of the lemmas often involve complete induction over a well-founded ordering. In proving such a lemma, we assume that the lemma holds for all elements of the well-founded set smaller (in the ordering) than a given element, and show that the lemma then holds for the given element as well.

In their paper, Manna and Waldinger present and illustrate the intermittent-assertion method with a variety of examples for proving total correctness. Some of their proofs are markedly simpler than their conventional counterparts. On the other hand, the

intermittent-assertion method is at least as powerful as the conventional invariant-assertion method and the well-founded ordering method, in addition to the more recent subgoal-assertion method for proving partial correctness.

The intermittent-assertion method not only serves as a valuable tool, but also provides a general framework encompassing a wide variety of techniques for the logical analysis of programs. Diverse methods for establishing partial correctness, termination, and equivalence fit easily within this framework. Furthermore, some proofs, naturally expressed with intermittent assertions, are not as easily conveyed by the more conventional methods. For example, the method can be applied to establish the validity of program transformations, and to prove the correctness of continuously operating programs, programs that are intended never to terminate.

This new method has begun to attract a good deal of attention. Different approaches to its formalization have been attempted, using predicate calculus, Hoare-style axiomatization, modal logic, and the Lucid formalism. It is believed that the intermittent-assertion method will have a practical impact on program verification, because it allows one to incorporate his intuitive understanding about the way a program works into a proof of its correctness.

3.1.2 Multiset Ordering Technique

A common tool for proving the termination of programs is the *well-founded set*, a set ordered in such a way as to admit no infinite decreasing sequences. The basic approach is to find a *termination function* that maps the elements of the program into some well-founded set, such that the value of the termination function is continually reduced throughout the computation. The well-founded sets most frequently used for this purpose are the natural numbers under the "greater-than" ordering and n-tuples of natural numbers under the lexicographic ordering.

All too often, the termination functions required are difficult to find and are of a complexity out of proportion to the program under consideration. However, by providing more sophisticated well-founded sets, the corresponding termination functions can be simplified. The goal of this research is to discover and apply suitable well-founded sets to the problem of termination.

Dershowitz and Manna [1978] have defined a class of well-founded orderings on multisets. *Multisets*, sometimes called *bags*, are like sets, but allow multiple occurrences of identical elements. For example, {3, 3, 3, 4, 0, 0} is a multiset of natural numbers; it is identical to the multiset {0, 3, 3, 0, 4, 3}, but is distinct from {3, 4, 0}.

The ordering on any given well-founded set S can be extended to form a well-founded ordering on the finite multisets over S. In this *multiset ordering*, a finite multiset M over S is greater than a multiset M', if M' may be obtained from M by the removal of at least one element from M and/or by the replacement of one or more elements in M with any finite number of elements taken from S, each of which is smaller than one of the replaced elements. Thus, if S is the set of natural numbers 0, 1, 2, . . . under the usual "greater-than" ordering, then the multiset {3, 3, 4, 0} is greater than each of the three multisets {3, 4}, {3, 2, 2, 1, 1, 1, 4, 0}, and {3, 3, 3, 3, 2, 2}. In the first case, two elements have been removed; in the second case, an occurrence of 3 has been replaced by two occurrences of 2 and three occurrences of 1; and in the third case, the element 4 has been replaced by two occurrences each of 3 and 2, and in addition the element 0 has been removed.

As an example of the use of a multiset ordering for a proof of termination, consider the following trivial program to empty a shunting yard of all trains:

```
loop until the shunting yard is empty
  select a train
  if the train consists of only a single car
     then remove it from the yard
     else split it into two shorter trains
     fi
  repeat .
```

This program is nondeterministic, as it does not indicate which train is to be selected nor how the train is to be split.

Let Y denote the set of trains in the yard, and $trains(Y)$ be the number of trains in the yard. For any train $t \in Y$, let $cars(t)$ be the number of cars it contains. We present two proofs of termination.

If we take the set of natural numbers as our well-founded set, then we are led to the selection of the termination function

$$\tau(Y) = 2 \cdot \sum_{t \in Y} cars(t) - trains(Y) .$$

This solution uses the fact that "splitting" conserves the number of cars in the yard, $\Sigma\, cars(t)$. Thus, splitting a train increases the number of trains in the yard, $trains(Y)$, by 1, thereby decreasing the current value of the termination function τ by 1. Removing a one-car train from the yard reduces $2 \cdot \Sigma\, cars(t)$ by 2 and increases $-trains(Y)$ by 1, thereby decreasing τ by 1.

If we use multisets of natural numbers as our well-founded set, then the function

$$\tau(Y) = \{cars(t) : t \in Y\}$$

demonstrates the termination of the shunting program. That is, for any configuration of the yard Y, $\tau(Y)$ denotes the multiset containing the size of each of the trains in Y. Each iteration of the program loop clearly decreases the value of $\tau(Y)$ under the multiset ordering: removing a train from the yard reduces the multiset by removing one element; splitting a train replaces one element with two smaller ones, corresponding to the two shorter trains.

The value of the multiset ordering is that it permits the use of relatively simple and intuitive termination functions in otherwise difficult termination proofs. In practice, using the more conventional orderings often leads to complex termination functions that are difficult to discover. For example, the termination proofs of programs involving stacks are often quite complicated and require much more subtle orderings and termination functions. Finding an appropriate ordering and termination function for such programs is a well-known challenge among researchers in the field of program verification. It is in this respect that the multiset ordering is of great help. We have, for example, used a multiset ordering to prove the termination of an iterative program to compute Ackermann's function. That proof is the most intuitive one known to us. Further research along these lines is under way.

We have found multiset orderings to be a particularly effective tool for proving the termination of iterative programs derived from recursive definitions, and for nondeterministic programs.

3.1.3 Verification of Production Systems

Programs are sometimes written in the form of a *production system*. There has been much recent interest in such systems for constructing symbolic simplifiers and theorem provers, and the problem of guaranteeing their correctness and termination is an actual one.

Consider the following production system, consisting of nine rewrite rules, intended to symbolically differentiates an expression with respect to x:

$$\begin{aligned}
&Dx \Rightarrow 1\\
&Dy \Rightarrow 0\\
&D(\alpha+\beta) \Rightarrow (\, D\alpha + D\beta \,)\\
&D(\alpha\cdot\beta) \Rightarrow (\, (\beta\cdot D\alpha) + (\alpha\cdot D\beta) \,)\\
&D(-\alpha) \Rightarrow (\, -D\alpha \,)\\
&D(\alpha-\beta) \Rightarrow (\, D\alpha - D\beta \,)
\end{aligned}$$

$$D(\alpha/\beta) \Rightarrow ((D\alpha/\beta) - ((\alpha \cdot D\beta)/(\beta\uparrow 2)))$$
$$D(ln\ \alpha) \Rightarrow (D\alpha / \alpha)$$
$$D(\alpha\uparrow\beta) \Rightarrow ((D\alpha \cdot (\beta \cdot (\alpha\uparrow(\beta - 1)))) + (((ln\ \alpha) \cdot D\beta) \cdot (\alpha\uparrow\beta))) ,$$

where y can be any constant or any variable other than x. Consider the expression

$$D(D(x \cdot x)+y) .$$

We could either apply the third production to the outer D, or else we could apply the fourth production to the inner D. In the latter case, we obtain

$$D(((x \cdot Dx)+(x \cdot Dx))+y) ,$$

which now contains three occurrences of D. At this point, we can still apply the third production to the outer D, or we could apply the first production to either one of the inner $D's$. Applying the third production yields

$$(D((x \cdot Dx)+(x \cdot Dx))+Dy) .$$

In general, at each stage in the computation there are many ways to proceed, and the choice is made nondeterministically. In our case, all choices eventually lead to the expression

$$((((1 \cdot 1)+(x \cdot 0))+((1 \cdot 1)+(x \cdot 0)))+0) ,$$

for which no further application of a production is possible.

The difficulty in proving the correctness of production systems stems from the fact that applying a production to a subexpression, not only affects the structure of that subexpression, but also changes the structure of its superexpressions, including the top-level expression. And a proof must take into consideration the many different possible sequences, generated by the nondeterministic choice of productions and subexpressions.

Proving the termination of a production system such as this one for differentiation is difficult, since some productions (the first two) may decrease the size of an expression, while other productions (the rest) may increase its size. Furthermore, a production (e.g. the fourth) may actually duplicate occurrences of subexpressions. (Manna and Ness [1970] describe a general method of proving the termination of production systems.)

An intuitive proof of termination of this system, using multisets, is based on the observation that the arguments to the operator D are reduced in size by each production. But since most of the productions increase the size of the expression as a whole, we need a termination function that takes the nested structure of the expression into consideration. We can do this by a natural extension of the multiset ordering to *nested multisets*. A nested multiset is either an element of some base set S, or else it is a finite multiset of nested multisets over S. For example,

$$\{\{1,\ 1\},\{\{0\},\ 1,\ 2\},\ 0\}$$

is a nested multiset. The *nested multiset ordering* is a recursive version of the simple multiset ordering: two elements of the base set S are compared using the ordering on S; any multiset is greater than any element of the base set; and two multisets are compared as in the simple multiset ordering.

So we let the well-founded set be the nested multisets over the natural numbers, and let the termination function yield the size of α for each occurrence of $D\alpha$, while preserving the nested structure of the expression. For example, the arguments of the six occurrences of D in the expression $D(D(Dx \cdot Dy)+Dy)/Dx$ are $D(Dx \cdot Dy)+Dy$, $Dx \cdot Dy$, x, y, y, and x. They are of sizes 9, 5, 1, 1, 1, and 1, respectively. Thus, for

$$e = D(\ D(\ Dx \cdot Dy\ ) + Dy\ )\ /\ Dx\ ) ,$$

we have

$$\tau(e) = \{\ \{9,\ \{5,\ \{1\},\ \{1\}\},\ \{1\}\},\ \{1\}\ \} .$$

3.1.4 References

Burstall, R.M. [Aug. 1974], *Program proving as hand simulation with a little induction*, Proc. IFIP Congress 74, Stockholm, pp. 308-312.

Dershowitz N. and Z. Manna [Mar. 1978], *Proving termination with multiset orderings*, Memo AIM-310, Stanford Artificial Intelligence Laboratory, Stanford, CA.

Manna, Z. and S. Ness [Jan. 1970], *On the termination of Markov algorithms*, Proc. 3rd Hawaii Intl. Conf. on System Sciences, Honolulu, HI, pp. 789-792.

Manna, Z. and R.J. Waldinger [Feb. 1978], *Is SOMETIME sometimes better than ALWAYS? Intermittent assertions in proving program correctness*, CACM, vol. 21, no. 2, pp. 159-172.

3.2 Proofs as Programs

Chris Goad

It often happens in mathematics that examination of a proof that a concrete mathematical object (e.g. a number) exists allows the explicit construction of that object. As an example, consider the following trivial theorem concerning the positive integers.

> For each x there exists a y such that y is divisible by each z which is less than x.

We prove the theorem by induction on x. For $x=1$, any choice of y will satisfy the theorem, since there is no positive integer less than 1 of which y must be a multiple. To be definite, we will take $y=1$. For the induction step, we assume the theorem holds at x, and prove that it holds for $x+1$. So assume there is a y divisible by each z less than x. Then $y \cdot x$ is divisible by each z less than $x+1$, so that the theorem holds for $x+1$.

If we have a particular value of x in mind, we can compute a corresponding y by "unwinding" the proof of the theorem in the obvious way – the number we get is $x-1$ factorial.

This kind of phenomenon was studied in detail by the mathematical logician Gentzen in the nineteen-thirties. He developed a mechanical method, which he called *normalization*, for unwinding proofs. Normalization has the important characteristic that, when applied to a "constructive" proof of the existence of a number having a certain property, it gives explicitly a particular number satisfying that property. Further, if the theorem proved has the form, "for all x there exists a y such that the property $\alpha(x, y)$ holds", then for each given number x normalization computes a number y satisfying $\alpha(x, y)$. It is in this sense that normalization allows one to treat proofs as programs. The class of proofs of formulas of the appropriate form can be regarded as the *set of programs of a programming language*, where normalization serves as the interpreter.

One can envision two possible applications to practical computing of the ability to treat a proof as a program: (1) direct application – the use of proofs as a programming language by humans, and (2) the use of a theorem prover for the automatic synthesis of computer programs. The first application depends for its usefulness on the differences between proofs and computer programs of the usual kind as expressions of computation. The second depends on the development of powerful automatic theorem provers. The work of Chris Goad, while relevant to (2), is motivated and directed primarily by (1). Specifically, he is investigating the nature and efficiency of proofs as expressions of computation, using both theoretical methods and computational experiments.

3.3 Program Testing and Debugging
Martin Brooks

The primary goal of Martin Brooks' (graduate student) research is to develop a theory of program testing. Such a theory tells one what inferences can be made from the observation that a program produces correct outputs on some finite set of inputs. Once such a theory is established its results will be used to build practical debugging tools to aid working programmers.

Program testing is the usual method by which programmers obtain confidence in the correctness of their programs. It is hoped among mathematicians and computer scientists that this will be replaced by automatic program verification. Programmers face two difficulties using the verification approach:

(1) The programmer must be able to precisely and *correctly* specify his program's intended behavior, and perhaps annotate his program, in some specification language.

(2) There is not much theory relating program incorrectness to difficulties in finding correctness proofs; verification theory does not address itself to debugging incorrect programs.

The goal is to tell how to automatically choose test examples; all the programmer must know is what his program's output *should* be on these examples.

This research reveals that for programs within certain general classes, called *debuggable classes*, there are algorithms, called *debugging algorithms*, which use program testing to either find or guarantee the nonexistence of certain types of programming errors, called *debuggable errors*. The theory allows for automatic construction of test inputs, so that a programmer need only be able to supply the correct outputs corresponding to the test inputs in order for the debugging algorithm to debug his program. If the programmer knows that the only possible errors he may have committed are debuggable errors, then correct outputs on the tests imply that his program is correct. If he has committed some debugable errors, his programs actual output will differ from its intended output on at least one of the automatically generated test inputs, and the debugging algorithm will correct the errors.

This theory of program testing leads to the possibility of a valuable programming tool: an automatic test case generator and debugger. This tool would have well understood properties and would be capable of demonstrating program correctness. It would be programmed to catch all instances of a finite number of kinds of debuggable errors. A programmer would use this tool as follows: First he writes his program and gives it to the debugger. The debugger analyzes the program, computes the appropriate inputs to test the program on, and then asks the programmer what the output should be on each of these inputs. The debugger compares the programmer's responses to the program's actual outputs. If they are the same then his program is guaranteed to be free of the sorts of bugs that the debugger is programmed to uncover. If some of his program's actual outputs are not correct, then the debugger uses them, sometimes after asking the programmer about some more test cases, to correct the programming errors which caused the bad outputs. Finally, the debugger returns the corrected version of the program to the programmer.

3.4 Program Synthesis
Zohar Manna

3.4.1 Overview

Program synthesis is the automatic construction of programs to meet given specifications. These specifications constitute a high-level description of the desired program, which expresses the purpose of the program, without indicating the method by which that purpose is to be achieved.

The specifications are expressed in terms of many constructs, which are endemic to the particular subject domain of the desired program, (e.g., numbers, sets, lists). Because these constructs are only intended to describe the purpose of the program and need not be computed, they can be of a much higher level than the constructs of any programming language (i.e., they can include logical quantifiers, set constructors, and other noncomputable operations). Thus, the specification language can correspond closely with the concepts a programmer actually uses in thinking about the problem.

The techniques we are developing are independent of the choice of a target programming language. The particular language we use in our examples and in our experimental system is a simple LISP-like language containing only basic numerical and list-processing operations, conditional expressions, and recursion. In considering the formation of programs with side-effects, we extend the language to include assignments to variables, array elements, and other data-structure components.

Our basic approach is to transform the specifications repeatedly according to certain rules; each rule replaces one segment of a program description by another, equivalent segment. The process continues until a description is obtained that is entirely in terms of the primitive constructs of the target language; this description is the desired program. The entire sequence of descriptions leading from the specifications to the final program is called a *program derivation*. The method guarantees that the final program will indeed satisfy the original specifications.

The transformation rules are guided by certain strategic controls which ensure that they are applied only at the appropriate time. Many of the transformation rules represent knowledge about the program's subject domain; some explicate the meaning of the constructs of the specification and target languages; a few rules correspond to basic programming principles, which are independent of the particular subject domain or programming language.

Some of the principles we have identified so far are:

- *Conditional formation* – This principles causes a case analysis to be introduced into the derivation, yielding a conditional test in the ultimate program.
- *Recursion formation* – This principle introduces a recursive call into the ultimate program by observing when a subgoal to be achieved is actually an instance of the desired top-level goal.
- *Well-founded ordering* – The termination of the recursive programs formed by the above technique is ensured by constructing a well-founded ordering with the property that the arguments of the program's recursive calls are all strictly less than the program's inputs.
- *Procedure formation* – A subsidiary procedure is formed when a subgoal is found to be an instance, not of the top-level goal, but of a previously generated subgoal.
- *Generalization* – A generalized procedure is formed when two subgoals are found to be an instance of a third expression, which is somewhat more general than both.
- *Simultaneous goals* – In constructing a program to achieve two or more goals simultaneously, we first construct a program to achieve one goal, then modify that program to achieve the others as well, while protecting the condition that was already achieved.

Further discussion of the same topics, at a more leisurely pace, along with bibliographical remarks and references, appears in the recent paper

Z. Manna and R. Waldinger, *Synthesis: Dreams => Programs*, Technical Report, Artificial Intelligence Lab., Stanford University, Stanford, CA (Nov. 1977). To appear in the CACM.

3.4.2 Basic Program Synthesis Techniques

Specifications

In designing the specification language, we have adopted many constructs (e.g., the set constructor or the logical quantifiers) that facilitate the description of a program but that cannot be included in the target programming language. We present below examples of specifications for simple programs using some of these high-level constructs.

A program *lessall*(x l), to test if a number x is less than every element of a list l of numbers, is specified as follows;

lessall(x l) ← *compute* $x < all(l)$
where *is a number and*
l *is a list of numbers.*

In general, the specification construct $P(all(l))$ denotes that the property P holds for every element of the list l.

The specification for a program to compute the greatest common division $gcd(x\ y)$ of two nonnegative integers x and y is

$gcd(x\ y)$ ← *compute max* $\{z : z|x$ *and* $z|y\}$
where x *and* y *are nonnegative*
integers and $x \neq 0$ *or* $y \neq 0$.

The set constructor $\{u : P(u)\}$ denotes the set of all elements u satisfying the property P.

The *all* construct $P(all(l))$ and the set construct $\{u : P(u)\}$ are nonprimitive specification constructs (i.e., they are not in the target programming language). The synthesis task is to transform a description of the desired program, such as the specifications presented above, into an equivalent description that employs only primitive constructs of the target language.

Transformation Rules

We use the notation

$$t \rightarrow t' \quad \text{if } P$$

to denote that a subexpression of form t may be replaced by the corresponding expression t', provided that the condition P is true.

For example, the rule

$$Q \text{ and } true \rightarrow Q$$

denotes the basic logical principle that an expression of form "Q *and true*" may be replaced by the simpler expression "Q". This rule has no conditions; it can always be applied.

The rule

$$P\ (all(l)) \rightarrow P(head\ (l))\ and\ P(all(tail(l))) \quad \text{if } not\ empty(l)$$

expresses the fact that a property P holds for every element of a nonempty list l, if it holds for the first element $head(l)$ and for every element of the list $tail(l)$ of the other elements. This rule imposes the condition that the list l be nonempty.

Derivation Trees

In developing a program whose specification are

$f(x)$ ← *compute* $P(x)$
where $Q(x)$,

we establish the output description as a goal to be achieved, viz.,

Goal: *compute* $P(x)$.

Subgoals are derived from this goal by application of the relevant transformation rules. For example, in deriving the *gcd* program, we form the top-level goal

Goal 1: *compute* $max\{ z : z|x \text{ and } z|y \}$.

By applying a transformation rule

$$y|v \text{ and } u|w \rightarrow u|v \text{ and } u|w-v$$

we obtain the subgoal

Goal 2: *compute* $max\{ z : z|x \text{ and } z|y-x \}$.

If a transformation rule imposes a condition P, which must be true for the rule to be applied, a subgoal of the form

Goal: *prove* P

must be achieved before the rule can be applied. For example, in developing the program *lessall*$(x\ l)$ to test if a number x is less than every element of a list l of numbers, we have the top-level goal

$$compute\ x < all(l),$$

which is obtained directly from the specification; in attempting to apply the rule

$$P(all(l)) \rightarrow true \quad \text{if } empty(l)$$

to this goal, we are led to the subgoal

Goal: *prove* $empty(l)$.

From each subgoal that is derived, further subgoals are generated by the application of more transformation rules. We thus construct a tree of goals and subgoals, which we will call a *program derivation tree*.

A subgoal "*compute* S" is already achieved if S consists entirely of primitive constructs of the target language. A subgoal "*prove* P" is achieved if P is the logical constant *true*. Such goals are terminal nodes of the derivation tree.

Our research so far has emphasized the identification and codification of the basic programming principles, those techniques that are applied again and again in the formation of a program, regardless of the particular subject domain. In the next few sections we discuss several of the basic programming principles which have been considered so far.

Conditional Formation

Many of the transformation rules impose conditions (e.g., l is nonempty, x is nonnegative) that must be satisfied for the rule to be applied. Suppose that in attempting to apply a particular rule, we fail to prove or disprove a condition P, where P is expressed entirely in terms of the primitive constructs of the programming language; in such a situation, the conditional-formation rule is invoked. This rule allows us to introduce a case analysis, and consider separately the case in which P is true and P is false. Suppose we succeed in constructing a program segment S_1 that solves our problem under the assumption that P is true, and another program segment S_2 that solves the problem under the assumption that P is false. Then the conditional-formation principle puts these two program segments together into a conditional expression

$$\text{if } P \text{ then } S_1 \text{ else } S_2,$$

which solves our problem regardless of whether P is true or false.

If we happen to generate the program segment S_2, say, without using the case assumption that P is false, then S_2 solves our problem regardless of whether P is true or false. In this case, no conditional expression is formed, and the program constructed is simply S_2. Thus, conditional expressions are generated only for truly relevant conditions.

The conditional-formation rule is among the best-understood of our basic programming principles.

Recursion-Formation and Well-Founded Ordering

Suppose, in constructing a program whose specifications are

$$f(x) \leftarrow \textit{compute } P(x) \textit{ where } Q(x)$$

we encounter a subgoal

$$\textit{compute } P(t)$$

which is an instance of our output specification, "*compute* $P(x)$." Because the program $f(x)$ is intended to compute $P(x)$ for any x satisfying its input specification $Q(x)$, the recursion-formation rule proposes achieving the subgoal by computing $P(t)$ with a recursive call $f(t)$. For this step to be valid, it must ensure that the *input condition* $Q(t)$ holds when the proposed recursive call is executed. To ensure that the new recursive call will not cause the program to loop indefinitely, the rule must also establish a *termination condition*, showing that the argument t is strictly less than the input x in some well-founded ordering. (A *well-founded ordering* is one in which no infinite strictly decreasing sequences can exist.) This condition precludes the possibility that an infinite sequence of recursive calls might occur during the execution of the program.

For example, the program *lessall*$(x\ l)$, which tests whether a given number x is less than every element of a given list l of numbers, was specified as follows:

$$\textit{lessall}(x\ l) \leftarrow \textit{compute } x < \textit{all}(l) \textit{ where } x \textit{ is a number and } l \textit{ is a list of numbers.}$$

In deriving this program, we develop a subgoal

$$\textit{compute } x < \textit{all}(\textit{tail}(l))$$

in the case that l is nonempty. This subgoal is an instance of our output specification, with the input l replaced by *tail*(l); therefore, the recursion-formation principle proposes that we achieve the subgoal by introducing a recursive call *lessall*$(x\ \textit{tail}(l))$. To ensure that this step is valid, the rule establishes an input condition, that

$$x \textit{ is a number and } \textit{tail}(l) \textit{ is a list of numbers,}$$

and a termination-condition that the argument pair $(x\ \textit{tail}(l))$ is less than the input pair $(x\ l)$ in some well-founded ordering. This termination condition holds because *tail*(l) is a proper sublist of l.

The recursion-formation principle is well-understood and has been applied together with the conditional-formation principle in the synthesis of many complete programs.

Procedure Formation

Suppose in developing a program whose specifications are of the form

$$f(x) \leftarrow \textit{compute } P(x) \textit{ where } Q(x)$$

we encounter a subgoal

Goal B: *compute* $R(t)$,

which is an instance, not of the output description "*compute* $P(x)$," but of some previously generated subgoal

Goal A: *compute* $R(x)$.

The procedure-formation principle proposes that we introduce a new procedure $g(x)$ whose output specification is

$$g(x) \leftarrow \textit{compute } R(x).$$

In this way, we can achieve both Goals A and B by calls $g(x)$ and $g(t)$ to a single procedure. In the case that Goal B has been derived from Goal A, the call to $g(t)$ will be a recursive call;

otherwise, both calls will be simple procedure calls.

For example, in constructing a program *cart(s t)* to compute the Cartesian product of two sets, we are given the specification

$$cart(s\ t) \leftarrow compute\ \{(x\ y) : x \in s\ and\ y \in t\}\ where\ s\ and\ t\ are\ finite\ sets.$$

In deriving the program, we obtain a subgoal

$$\textbf{Goal A:}\ compute\ \{(x\ y) : x = head(s)\ and\ y \in t\}$$

in the case that s is nonempty. Developing Goal A further, we derive the subgoal

$$\textbf{Goal B:}\ compute\ \{(x\ y) : x = head(s)\ and\ y \in tail(t)\}$$

in the case that t is nonempty. Goal B is an instance of Goal A; therefore, the procedure-formation rule proposes introducing a new procedure *carthead(s t)* whose output specification is

$$carthead(s\ t) \leftarrow compute\ \{(x\ y) : x = head(s)\ and\ y \in t\}$$

so that we can achieve Goal A with a procedure call *carthead(s t)* and Goal B with a (recursive) call *carthead(s tail(t))*.

Our method for proving the termination of ordinary recursive calls does not always extend to the multiple-procedure case.

Generalization

Suppose in deriving a program we obtain two subgoals

$$\textbf{Goal A:}\ compute\ R(a(x))$$

and

$$\textbf{Goal B:}\ compute\ R(b(x)),$$

neither of which is an instance of the other, but both of which are instances of the more general expression

$$compute\ R(y).$$

Then the extended procedure-formation rule proposes that we introduce a new procedure, whose output specification is

$$g(y) \leftarrow compute\ R(y),$$

so that we will be able to satisfy Goal A by a procedure call $g(a(x))$ and Goal B by a procedure call $g(b(x))$.

For example, in constructing a program to reverse a list, we derive two subgoals

$$\textbf{Goal A:}\ compute\ append(reverse(tail(l))\ cons(head(l)\ nil))\ .$$

$$\textbf{Goal B:}\ compute\ append(reverse(tail(tail(l)))\ cons(head(tail(l))\ cons(head(l)\ nil))).$$

Each of these goals is an instance of the more general expression

$$compute\ append(reverse(tail(l))\ cons(head(l)\ m))\ ;$$

therefore, the extended procedure-formation rule proposes introducing a new procedure *reversegen(l m)*, whose output specification is

$$reversegen(l\ m) \leftarrow compute\ append(reverse(tail(l))\ cons(head(l)\ m))\ .$$

This procedure reverses a nonempty list l and appends the result to m. Although the procedure solves a more general problem than the *reverse* program we actually require, it turns out that the *reversegen* procedure is actually easier to construct.

3.4.3 Structure-changing Programs

In the discussion so far, we have been concerned with structure-maintaining (i.e., "side-effect-free") programs, which produce no permanent change in the data objects of the programming environment. The same principles apply to the development of structure-changing programs, which can produce such changes. However, certain new problems arise in the synthesis of structure-changing programs; among these is the *simultaneous-goal problem*.

In constructing a program to achieve two conditions P_1 and P_2, it is not sufficient to decompose the problem by constructing two independent programs to achieve P_1 and P_2, respectively. The program that achieves P_2 may in the process make P_1 false, and vice versa. Thus, the concatenation of the two programs will not achieve both conditions.

For example, suppose we want to construct a program to sort the values of three variables x, y, and z; in other words, we want to permute the values of the variables to achieve the two conditions $x \le y$ and $y \le z$ simultaneously. Assume that we are given the primitive instruction *sort2*$(u\ v)$, which sorts the values of its input variables u and v. Then we can achieve each of our desired conditions independently by executing the program segment *sort2*$(x\ y)$ and *sort2*$(y\ z)$, respectively. However, the concatenation

sort2$(x\ y)$
sort2$(y\ z)$

of these two segments will not achieve both conditions simultaneously; in sorting y and z, the second segment *sort2*$(y\ z)$ may make the first condition $x \le y$ false.

To circumvent difficulties of this sort, we have introduced the following simultaneous-goal principle:

To satisfy a goal of form

achieve P_1 *and* P_2,

first construct a program F to achieve P_1, then modify F to achieve P_2 while protecting the truth of P_1 at the end of F. The program-modification technique we employ is based on the "weakest-precondition operator." A special "protection mechanism" ensures that no modification is permitted that destroys the truth of the protected condition P_1 at the end of the program.

To apply this principle to the goal

achieve $x \le y$ *and* $y \le z$

in the sorting problem, we first construct the program segment *sort2*$(x\ y)$ that achieves the first condition. We then modify this program to achieve the second condition $y \le z$. We cannot achieve this condition by inserting the instruction *sort2*$(y\ z)$ at the end of the program, because (as we have seen) this modification violates the condition $x \le y$, which we must protect.

However, our program-modification technique allows us to achieve a goal by inserting modifications at any point in the program, not merely at the end. In this case, the technique causes us to introduce the two instructions

if $y < x$ then *sort2*$(x\ z)$

and

if $x \le y$ then *sort2*$(y\ z)$

at the beginning of the program segment. The modified program

if $y < x$ then *sort2*$(x\ z)$
if $x \le y$ then *sort2*$(y\ z)$
sort2$(x\ y)$

will achieve both conditions $x \le y$ and $y \le z$ simultaneously.

The derivation of straight-line programs with

simple side-effects is now fairly well understood; much work needs to be done on the derivation of structure-changing programs with conditional expressions and loops, and the derivation of programs that alter list structures and other complex data objects.

3.4.4 Applications to Programming Methodology

Although the development of a practical program-synthesis system requires considerable research effort, certain applications of program-synthesis techniques to more restricted problems will be of more immediate practical value. Let us consider several of these areas, to see where program-synthesis techniques may be applicable.

Structured Programming – Like program synthesis, structured programming presents principles for deriving a program systematically from given specifications. However, the principles of structured programming are intended to guide a human programmer, whereas the principles of program synthesis are meant to direct a computer system. Nevertheless, we have found that some of the techniques we have developed for a program-synthesis system could well be employed by a human programmer. In particular, we show that the recursion-formation principle is a better motivated guide for introducing a loop than the conventional structured-programming method for the same task.

Program Transformation – In this approach, the programmer constructs a transparent program for his task, which is likely to be correct but which may be inefficient. This program is then transformed into an efficient equivalent program, which may be more difficult to understand. This transformation process is guaranteed to produce a program equivalent to the original. Program transformation may be regarded as a synthesis task in which the specifications are given in the form of a clear program in the target language. All the synthesis techniques we have developed can be applied to program transformation as well.

Data Abstraction – In this approach, the programmer expresses his program in terms of *abstract data types*, objects such as sets, queues, or graphs whose properties are well-defined but whose precise machine representation is left unspecified. When this program is complete, representations for its abstract data types are chosen and the program is transformed to replace the operations on the abstract data types by the corresponding concrete operations on the chosen representation. Program-synthesis techniques can be applied to perform this transformation process.

Program Modification – It is often observed that programmers spend more of their time extending programs that already perform some task correctly than they do in developing new programs. This process is particularly fraught with error, because in modifying a program, the programmer is likely to make some change that interferes with the program's original functioning. We have remarked that a program-modification technique was developed to support the simultaneous-goal principle. This technique can also be applied to perform independent program-modification tasks. The protection mechanism ensures that the modified program must still perform the task for which it was originally intended.

3.4.5 Related Publications

Our work on program synthesis, which was partially supported by the ARPA Contract to the Stanford Artificial Intelligence Laboratory, resulted in the following publications:

1. Z. Manna and R. Waldinger, *Towards automatic program synthesis*, CACM, Vol. 14, No. 3 (March 1971), pp. 151–165,

2. Z. Manna and R. Waldinger, *Knowledge*

and reasoning in program synthesis, Artificial Intelligence, Vol. 6, No.2, pp. 175-208,

3. N. Dershowitz and Z. Manna, *The evolution of programs: A system for automatic program modification*, IEEE Transactions on Software Engineering, Vol. 3, No. 5 (Nov. 1977), pp. 377-385,

4. Z. Manna and R. Waldinger, *The logic of computer programming*, IEEE Transactions on Software Engineering, Vol. SE-4, No. 5 (May 1978),

5. Z. Manna and R. Waldinger, *Synthesis: Dreams => programs*, CACM (to appear),

and in the following conference presentations:

6. N. Dershowitz and Z. Manna, *On automating structured programming*, Proceedings of the International Symposium on Proving and Improving Programs, Arc-et-Senans, France (July 1975), pp. 167-193,

7. Z. Manna and R. Waldinger, *The automatic synthesis of recursive programs*, Proceedings of the Symposium on Artificial Intelligence and Programming Languages, Rochester, NY (Aug. 1977), pp. 29-36,

8. Z. Manna and R. Waldinger, *The automatic synthesis of systems of recursive programs*, Proceedings of the Fifth International Joint Conference on Artificial Intelligence, Cambridge, MA (Aug. 1977) pp. 405-411,

9. Z. Manna and R. Waldinger, *The synthesis of structure-changing programs*, Proceedings of the Third International Conference on Software Engineering, Atlanta, GA (May 1978),

10. Z. Manna and R. Waldinger, *The DEDALUS system*, Proceedings of the ACM National Computer Conference, Anaheim, CA (June 1978).

4. Program Verification

Personnel: David Luckham, Derek Oppen, Friedrich vonHenke, *Student Research Assistants*: Richard Karp, Wolfgang Polak, William Scherlis.

The research of the Program Verification Group is directed towards the development of new programming methods and automated programming aids. The goal is efficient production of very reliable systems programs including compilers and operating systems, and efficient maintenance of such programs.

The group is actively pursuing three main research areas:

1. Design and implementation of interactive program analyzers.

2. Design of a high level programming language and associated documentation language for concurrent processing.

3. Application of program analyzers, and particularly verifiers, to such programming problems as debugging, documentation, proof of correctness, and analysis of modifications to code and specifications.

Systems that automate or partially automate the analysis of properties of programs may be collectively named "program analyzers". The group has implemented two analyzers, the Stanford Verifier and the Runcheck system. The current Stanford Verifier automates methods for analysing the consistency of a program with its documentation. It is already a useful debugging tool. The Runcheck version of our verifier, is designed to analyze a program for possible common runtime errors. It automates some simple methods for improving documentation and analysing why verifications fail.

Our experiments with these analyzers have made it clear that verification methods can be applied to analysis of other programming problems, including adequacy of documentation, efficiency of code, and adaptability of existing code to new specifications.

Currently the group is working towards applying analysis based on verification techniques to a very wide range of programs. These include:
(i). new kinds of programs previously considered to be beyond the limits of our techniques; e.g., complicated pointer manipulating programs, and large programs such as a compiler.
(ii). programs using new language constructs such as Modules and Concurrent Processes.

This has required a research effort in design of programming languages and documentation languages, and in programming methods, particularly in the area of concurrent processes. This effort has been carried out with very careful attention to the DoD Common High Level Programming Language specifications [3].

The references contain some of the group's earlier work in areas 1 and 3 (above) which has been published (e.g., [2, 7, 10, 11, 15, 18, 19, 27]). An overview of this work is contained in [16].

4.1 Accomplishments

1. The group has implemented a verifier for almost the full Pascal language (exclusions mainly concern floating point arithmetic). The user manual [26] contains an introduction to program verification and many examples illustrating the use of the verifier as an aid in debugging, documentation and structured programming. The verifier is currently being introduced at two other ARPAnet sites to test its portabilty and to obtain some preliminary feedback from other user groups. Distribution on a limited basis is planned for Fall 1978. A library of documented and verified programs is being built and is available over the ARPAnet from SAIL on the directory [EX,VER].

2. A special version of this verifier for automatic detection of runtime errors in programs has been implemented. This is called the Runcheck system. Results with an early version of Runcheck have been published (German, [6]), and are the most impressive in the area of completely automatic analysis of programs so far. Work on improvements to this system and on automatic documentation of Pascal programs is continuing.

3. The success of this verifier depends on recent advances made by this group in the theory and implementation of cooperating special purpose decision procedures (Nelson and Oppen [20, 21], Oppen [22]). This new approach to implementing theorem provers is the best method found to date. It will doubtless play an important role in the implementation of sophisticated analysis and decision programs in application areas other than program verification.

In developing an underlying theorem proving capability a second important area of progress has been the design and implementation of a Rule Language for user-supplied lemmas. This latter facility allows the user to describe data structures not handled by special purpose provers and to define new concepts used in the specification of programs. The rule language is a facility for defining specification languages. Its implementation gives the verifier the ability to use definitions of specification concepts in correctness proofs. Reports on this facility are in the user manual [26] and in [24].

4. The language accepted by the verifier extends Pascal by inclusion of Modules in a form compatible with the DoD Ironman specifications on Encapsulation (DoD Ironman [3]). A theory of documentation of modules has been worked out (Luckham and Polak [17]). This theory is new and has not been suggested in any other previous language design. Its practicality is being tested. The verifier requires modules to be documented. Verification of documented modules has already been successful on some substantial examples. One example is a module implementing Pascal pointer and memory allocation operations within a subset of Pascal whose only complex data type is Arrays.

5. It is proposed that the programming language accepted by the verifier include features for concurrent processing. A specific design based on previous work of Kahn [13,14], Brinch Hansen [1], and Wirth [28], is being considered. This design satisfies most of the DoD Ironman design specifications for parallel processes ([3] section 9). A simple operating system has been written in our language and is being studied. A theory of semantics of concurrent processes and methods of documenting them is being worked out and tested on parts of the operating system.

4.2 References

[1] Brinch-Hansen, P., "The Solo Operating System: A Concurrent Pascal Program", Software – Practice and Experience 6 (1976).

[2] Cartwright, R. and Oppen, D., "Unrestricted Procedure Calls in Hoare's Logic," Proceedings of the Fifth ACM Symposium on Principles of Programming Languages, ACM, New York (1978).

[3] DoD., "Requirements for High Order Computer Programming Languages", Department of Defense "revised Ironman", July 1977.

[4] Drysdale, R.L., "Verification of sorting programs", AI Lab. Memo, forthcoming.

[5] Flon, L. and Suzuki, N., "Nondeterminism and the Correctness of Parallel Programs," Proc. IFIP Working Conference on the Format Description of Programming Concepts, St. Andrews, New Brunswick (1977).

[6] German, S., "Automating Proofs of the Absence of Common Runtime Errors", Proceedings of the Fifth ACM Symposium on Principles of Programming Languages, ACM, New York pp.105-118, (1978).

[7] F.W. v.Henke, and D.C. Luckham, "A methodology for verifying programs", Proc. Int. Conf. on Reliable Software, Los Angeles, California, April 20-24, 1975, 156-164.

[8] v. Henke, F.W. and Luckham, D.C., "Verification as a Programming Tool", A.I. Lab. Memo, forthcoming.

[9] Hoare, C.A.R., "Monitors: An Operating System Structuring Concept", CACM 17: 10 (1974), pp 549 – 557.

[10] G. Huet, D.C. Luckham, and D. Oppen, Proving the absence of common runtime errors, Draft, Stanford A.I. Lab. Memo, Stanford University, forthcoming.

[11] S. Igarashi, R.L.London, and D.C. Luckham, Automatic program verification I: Logical basis and its implementation, *Acta Informatica*, vol.4, 1975, 145-182.

[12] Jensen, K. and Wirth, N., "Pascal User Manual and Report," Springer-Verlag, New York (1975).

[13] Kahn, G., "The Semantics of a Simple Language for Parallel Programming," Proceedings of IFIP, North-Holland Publishing Company, Amsterdam (1974).

[14] Kahn, G. and MacQueen, D., "Coroutines and Networks of Parallel Processes," Proceedings of IFIP, North-Holland Publishing Company, Amsterdam (1977).

[15] Karp, R., and Luckham, D., "Verification of Fairness in an Implementation of Monitors," 2nd Intl Conf on Software Engineering, San Francisco (1976).

[16] Luckham, D.C. "Program Verification and Verification-Oriented Programming", Proc. IFIP Congress 77, pp. 783-793, North-Holland publishing Co., Amsterdam, Aug. 1977.

[17] Luckham, D. and Polak, W., "Verification in Languages with Modules", draft manuscript, A.I. Lab. Memo, forthcoming, submitted for publication (1978),

[18] D.C. Luckham, and N. Suzuki, Automatic program verification IV: Proof of termination within a weak logic of programs, AIM-269, Stanford Artificial Intelligence Project, Stanford University, 1975, *Acta Informatica*, vol. 8, 1977, 21-36.

[19] D.C. Luckham, and N. Suzuki, "Verification oriented proof rules for arrays, records, and pointers", Stanford Artificial Intelligence Project Memo 278, Stanford University, April 1976; revised for publication: "Verification of Array, Record, and Pointer Operations in Pascal", June 1978.

[20] Nelson, C. and Oppen, D., "Fast Decision Procedures based on Congruence Closure", Memo AIM-309, CS Report No. STAN-CS-77-646, Stanford University (1978); also, Proceedings of the 18th Annual IEEE Symposium on Foundations of Computer Science (1977).

[21] Nelson, C. and Oppen, D., "Simplification by Cooperating Decision Procedures", AI Memo AIM311, CS Report No. STAN-CS-78-652, Stanford University; also, Proceedings of the Fifth ACM Symposium on Principles of Programming Languages, ACM, New York (1978).

[22] Oppen, D., "Reasoning about Recursively Defined Data Structures", Proceedings of the Fifth ACM Symposium on Principles of Programming Languages, ACM, New York (1978).

[23] Owicki, S., "Specifications and Proofs for Abstract Data Types in Concurrent Programs," Technical report 133, Digital Systems Laboratory, Stanford, 1977.

[24] Scherlis, W., "A Rule Language and its Implementation", in preparation, A.I. Lab Memo, forthcoming.

[25] Scott, D., "Outline of a Mathematical Theory of Computation," Technical monograph PRG-2, Oxford University Computing Laboratory, Oxford, England (1970).

[26] Stanford Verification Group, Ed. Oppen, "Stanford Pascal Verifier User Manual", in preparation, A.I. Lab. memo, forthcoming.

[27] Suzuki, N., "Automatic Verification of programs with complex data structures," Ph.D. thesis, computer science dept., Stanford University, Stanford, Ca (1976).

[28] Wirth, N., "MODULA: A language for modular multiprogramming", ETH Zurich, 1976.

5. Image Understanding

Personnel: Thomas Binford, *Student Research Assistants*: Donald Gennery, Reginald Arnold, Rodney Brooks, Russell Greiner

The objective of this research is to solve scientific problems that are crucial in accomplishing certain tasks in photointerpretation and guidance. The two areas of research considered here are algorithms for shape representation and shape matching and algorithms for stereo mapping. A further objective is to incorporate these algorithms in computer systems which monitor airfields and buildings, and which locate airfields, aircraft, and vehicles in aerial photos.

5.1 Introduction

A photointerpreter typically works with a sequence of images taken from an aircraft or satellite. He solves a puzzle by piecing together selected and multiple clues from current images, background information, and previous images. In doing so, he relies heavily on spatial interpretation from stereo imaging and shadows, and spatial knowledge about structures. Cartographers produce contour maps from stereo images and maps of special features (e.g. roads) and incorporate them in digital data bases. Photointerpretation (PI) and cartography production systems are being increasingly automated using computers. The payoff is quicker response, higher throughput, and increased availability of resulting data.

The results of this research are expected to provide for automated systems the spatial interpretation and effective stereo mapping capabilities which are important for human performance. These facilities will help to enhance the power of interactive aids in measuring, counting, and recording, and in screening and monitoring.

Cruise missile guidance systems rely on radar range data matched against terrain elevation data. Stereo and motion parallax ranging is attractive for guidance because it is passive. Stereo ranging can be very accurate and has high spatial resolution. It makes use of well-developed sensors. Results of this program of research would facilitate greater survivability through use of passive sensing. At low altitude where cruise missiles and tactical missiles operate, the world appears intrinsically three-dimensional; this research should lead to useful capabilities in three-dimensional sensing, navigation, and terrain modeling.

The major objective of this research is to solve scientific problems which stand in the way of solution of practical problems in PI and guidance. A part of their solution is implementation of algorithms and systems which provide new image understanding capabilities and which are generalizable in the following sense: algorithms should solve well-defined subproblems which are common to typical tasks; systems for different tasks should be assembled from a core of common modules and a few modules which are specific to the task. This objective has been approached by carrying out sample tasks in PI and guidance in designing and implementing the model-based system. Instead of using special-purpose routines, the general purpose modeling and matching system will be used. The same system should be useful for vehicles, aircraft, and airfields.

The following scenarios indicate potential uses of these capabilities:

Scenario I

Interactive aids to measurement: a PI system measures capacity of oil tanks, cargo capacity of truck traffic, and sizes of buildings indicated by an interpreter.

Scenario II

An interpreter instructs a PI system to monitor aircraft on selected airfields. The system has a library containing generic models of classes of aircraft and specific aircraft models. For example, it has a model of the class of commercial jet passenger aircraft, and models of Lockheed L-1011, Douglas DC-10, and Boeing 747. The interpreter builds models of the airfields by indicating runways, taxiways, parking areas and passenger terminals. The interactive system helps by grouping edges and surfaces and making measurements. Airfield models go into a library for future use. As images come in, the system locates the airfield, uses stereo to map the airfield, pick out objects on the field, and identify those which are aircraft by class and type if known.

Scenario III

A building complex is monitored for changes. The system gives notice when new structures are found, or roads or nearby airfields are changed. The interpreter makes an assessment. The system aids him in updating and recording the model of the complex.

Scenario IV

An area is monitored for new buildings, new airfields, and new roads.

The tasks in these scenarios require operations of counting, measuring, and comparing. To count requires identification. To measure requires segmenting features. For example, measuring the volume of an oil tank requires separating out the top, measuring its diameter, separating the vertical cylinder and measuring its height. To compare and to identify require a mechanism to build models of the buildings or objects to be compared and descriptions of meaningful differences. Programming new tasks should be feasible for photointerpreters and not require months of effort by experts in computer vision. Carrying out the tasks in these scenarios makes use of three image understanding abilities:

a. stereo mapping and segmented description,
b. shape matching and modeling, and
c. a system for programming PI tasks.

Spatial Understanding

The approach of this research is based on a few observations.

> Typical data which are desired in PI tasks are identification of objects, measurement of their location and orientation, and description of their spatial structure and spatial relations.
>
> Most forms of collateral knowledge, a priori knowledge, and world knowledge are knowledge about objects, surfaces, and spatial relations.
>
> A natural means for photointerpreters to specify PI tasks is in terms of spatial models of objects and relations.
>
> Most low level constraints on stereo mapping and segmentation are geometric relations in image and spatial domains.
>
> Recognition is most simple and most general when image elements are interpreted as three-dimensional spatial structures.

The approach adopted here was to perform most interpretation in the spatial (three-dimensional) structural domain, to build powerful generic internal modeling capabilities for shape and spatial relations, and to build effective stereo spatial perception. This approach to image understanding has been called *spatial understanding*.

5.1.1 Stereo Mapping and Segmentation

Research has been directed toward doing stereo mapping better and doing it faster. The mapping operation produces a three-dimensional map of visible surfaces of buildings and terrain, together with a segmentation and symbolic description of

surfaces and their relations. One form of the map is a table z(x,y) of height as a function of image coordinates, with a symbolic data base of surfaces described as plane or cylindrical, with orientations horizontal, vertical, or otherwise, and with parallel and orthogonal relations between surfaces. The ground surface may be delineated and there may be additional terrain modeling.

Consider what is desired of the stereo mapping process. It should be possible to make measurements and segmentations of surfaces which are as accurate as the image data permits, over selected parts of images. It is not necessarily desirable to do so for entire images. The mapping algorithm should be efficient; that is it should perform in near-minimum time for a given level of performance and given level of hardware technology.

It should be possible to choose a good tradeoff of performance and system complexity. To do so, it is important to characterize and parameterize classes of mapping algorithms and to evaluate their performance and complexity. All of this requires a comprehensive analysis of stereo mapping, an objective of this research.

In cartography, a typical function is mapping elevation contours. Several systems have partial success with the simple case of mapping smooth terrain. However, they have problems at buildings and with thin objects (surface discontinuities), over water and pavement which are uniform or surfaces with repetitive markings (ambiguity problem), and in trees where there are a range of elevations. The systems have to be started manually, The most success in dealing with these problems has been achieved here at SAIL and at CDC. These programs are complementary and still in progress. Fundamental work on analysis of stereo has been carried out at MIT AI Lab.

Research on advanced stereo mapping systems is progressing rapidly. The chief scientific problems to be solved are: (1) automating the matching of corresponding parts of images at surface discontinuities; (2) resolving ambiguities by using global correspondence; and (3) designing algorithms and machine architectures to meet time objectives.

This research program has taken the following approaches to solving those problems: (a) development of edge-based stereo which deals with the problem of surface discontinuity and provides increased accuracy, resolution and speed; (b) analyses of stereo matching which provide a fundamental basis for algorithms for mapping; (c) interpretation in terms of surfaces which aids in cutting computation and resolving ambiguities. The research would contribute to measurement by segmenting surfaces regardless of surface markings and camouflage. Intrinsic three-dimensional measurements enable accurate measurement of non-planar objects.

The current level of performance of stereo mapping systems here is described in the proceedings of recent Image Understanding Workshops [Arnold, Gennery]. Stereo systems produce depth maps which appear adequate for interpretation by the model-based PI system. Both area-based and edge-based stereo systems have been implemented. The output of these programs is a three-dimensional map of small areas in one case, edges in the other. The ground plane and large planes are described and their boundaries are crudely delineated.

Existing change monitoring systems use differencing of pairs of images after extensive registration and warping. In radar images, differencing may be effective if the images are taken from the same viewpoint. In the visible part of the spectrum, changes in sun angle and shadows, rain or snow, and clouds cause intensity differences but these are not interesting changes. Meaningful changes are changes in spatial structures, e.g. new buildings.

In the approach used here, a three-dimensional structural description of the site is constructed with the aid of existing knowledge of the site. Structural differences are reported. The model-based system and the stereo mapping system are used together in this case.

5.1.2 Model-Based PI System

The second topic of this research is the design and implementation of a model-based PI system. An interactive PI system should aid an interpreter by automating routine tasks like counting oil tanks, measuring their volume, and recording the results, and by taking over low priority screening and monitoring tasks. To perform these tasks, the system must be able to discriminate the objects to count or monitor, and must delineate boundaries of surfaces whose dimensions are to be measured. In simple cases, objects can be discriminated primarily by position obtained from maps; trains run on rails; ships are found at piers. However, a typical PI task is to monitor traffic of manpower, arms, and supply in one area. To carry out this task, the PI system needs to identify tanks and military vehicles, trucks, gasoline trucks, and personnel carriers and to identify types of railroad cars and ships with their cargos. These tasks require an ability to describe shape and discriminate on the basis of shape.

Shape Matching

Previous research here at SAIL has demonstrated effective three-dimensional matching of complex shapes [Nevatia]. Alternative techniques using the matching of moments of orthogonal polynomials [McGhee] identify aircraft from complete and perfect silhouettes of isolated aircraft. This is a two-dimensional approach. An exhaustive set of projected views of each aircraft is recorded, and the best match is found. That approach has only limited utility. It is not now feasible to get perfect silhouettes of isolated objects reliably. For instance, aircraft are often not isolated, but are connected to passenger ramps and service vehicles. Further, such techniques are not capable of generic matching. That is, an aircraft with external pods cannot be recognized as a variant of the unmodified craft.

Several systems are relevant to counting and recording in interactive PI systems. SRI has demonstrated a module which counts railroad boxcars and another which could potentially be used for counting vehicles on a road. Rochester has shown how ships at piers can be counted. In this approach, little shape information is used. The systems make use of the restricted context of a road or rails or a pier. On rails, only railroad cars are expected, and it is only necessary to measure the beginning and ends of cars. If it is necessary to determine types of vehicles on roads or cars on rails, then more detailed shape discrimination is essential. In the tasks described in the preceding scenarios, powerful shape description and matching mechanisms are important.

The approach used here has the capability for generic interpretation and for detailed shape discrimination. It is a three-dimensional approach. The proposed approach does not need a complete or perfect segmentation. It integrates segmentation with identification and uses local shape elements for cues to further detailed matching. It can thus tolerate many errors, as long as there are significant local features correctly described. Use is made of three-dimensional data and cues, including internal edges and markings and stereo maps. Its representation of shape is superior, which means that it can make better discrimination, with more compact representations. It has a three-dimensional part/whole representation which allows more accurate description of parts than other techniques.

For example, this representation includes the knowledge that wings and tail are laminar, while the two-dimensional silhouette matching technique has no separate concept of wing or tail and no notion of three-dimensional shape.

The representation enables the description of objects by generic parts, which makes possible generic interpretation.

Programming PI tasks

Three systems represent the state of the art. The first, the Verification Vision system, was built in research here at SAIL [Bolles]. It depends on matching parts of a training image to the test image. The orientation of objects in the test image must be approximately the same as in the training image. It has the ability to order the sequence of operations in a cost-effective way. It would be effective for many problems, but not in the scenarios being investigated here. For example, we do not know in advance how aircraft will be oriented. It has no capability for generic matching and generic models. Its only primitives are matching correlation patches and curve matching.

The second, the Hawkeye system of SRI [Barrow], addresses tasks which can be simplified by use of a digital terrain map and a road map. It uses a detailed map data base of digital elevation data and a road data base. The system provides capabilities for registering images with the digital terrain map and localization of points of interest. With these facilities, it has carried out several road and rail monitoring tasks which are tightly constrained to one-dimensional searches along rails or road.

The third, the ACRONYM system, is being built here at SAIL to deal with the issues raised in programming PI tasks. One of these key requirements is generic modeling and interpretation. Another key issue is encoding and using knowledge of PI experts. PI tasks typically require the integration of knowledge from structural interpretations of images with knowledge from collateral sources.

Ultimately, interpreters should program in natural language. There are rudimentary natural language systems which could be used. A system like ACRONYM is the missing link. That is, natural language is useful only when it is very high level language. Using natural language to program at the level of FORTRAN, LISP, or SAIL is of little help. The hard part in writing vision programs is programming vision-specific problems, not writing FORTRAN. Using natural language interfaced to ACRONYM may be interesting.

5.2 Progress in Model-Based Interpretation

The design criteria for the model-based system are: the same system should be used for several different tasks with minor modifications; the system should be capable of generic interpretation, e.g. identifying an object as an aircraft without necessarily identifying it as an L-1011 or DC-10; an interpreter should find it easy to specify a new task. The system is being built with airfields, oiltanks, aircraft, buildings, and vehicles as examples for interpretation and measurement. The system uses models in a more powerful way than other approaches and is expected to lead to more powerful and more robust performance in monitoring, measuring, and counting. The design and much of the implementation are completed. The following paragraphs indicate how these design objectives were met.

A natural way for an interpreter to specify a task is in terms of object models, rather than programs. The system was designed with a high level modeling language. A convenient common language for interpreter and system is based on object models.

The criterion of generic interpretation forces an approach different from other recognition systems which are oriented toward recognizing specific instances, not class interpretation. The solution which was chosen was to use a part/whole representation in terms of generic parts. The generalized-cone representation was originally designed to make this possible. The key design requirement was that the primitives in the volume representation must

aid in generic description of parts of objects. For example, the representation should lead to description of a fuselage as a circular cylinder and wings as planar surfaces.

The interpretation process chosen was to match elements of generalized cones with elements of line drawings or surface elements of surface maps, and to match relations between generalized cones with relations between elements of line drawings or surfaces. That process could go either top-down or bottom-up. That is, it could also match elements of line drawings or surface maps with elements of generalized cones. An original intention of the generalized-cone representation was that three-dimensional cones map in a natural way to two-dimensional cones (ribbons), and that two-dimensional cones have a natural interpretation in terms of three-dimensional cones.

A subsystem was designed for predicting appearances of parts of objects and one for planning the strategy of matching. If the predictor were intended to predict appearances for specific objects from specific viewpoints with gray scale output, it could be done by standard graphics techniques. Instead, it is intended to predict appearances for classes of objects over a range of viewpoints with symbolic output. The predictor is intended to choose quasi-invariants (those features which are approximately constant over a wide range of viewpoints and parts variation).

Thus, there are three parts to the model-based PI system: the high level modeling subsystem, whose output is an Object Graph; the predictor and planner whose output is an Observability Graph; and the matcher whose output is an Interpretation Graph. A schematic of the system is shown in figure 1.

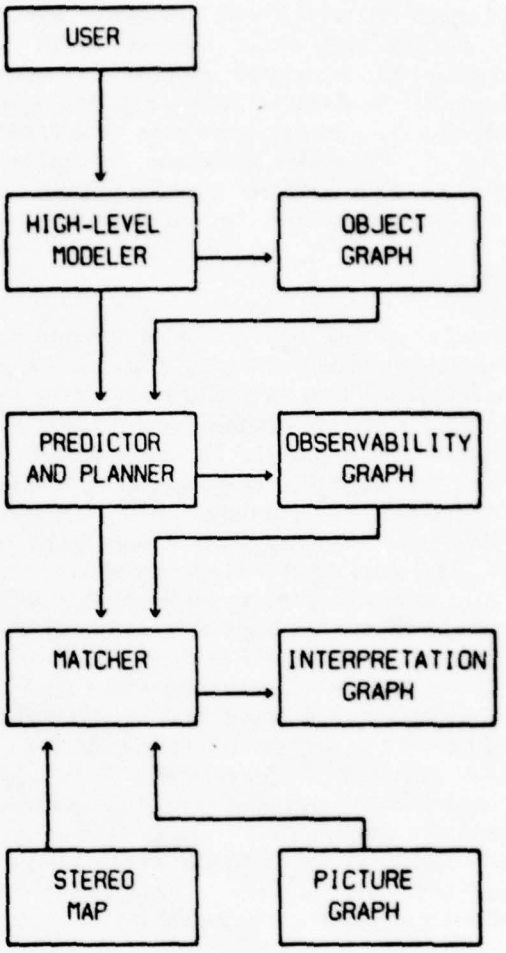


Figure 1: Model-Based PI System

In a typical scenario, a photointerpreter gives a brief symbolic description of a typical airfield and shows examples of some airfields, from which specific and generic properties are inferred. Initially, the most profound inferences must be made by the user. That is, he must specify initially which properties and dimensions are criterial, that is, necessary, and intrinsic, and not just an accident of the examples. It is expected that some of these inferences can be made automatically, but usually a broad knowledge base of functional design is required to make such inferences

automatically. For a simple example, to infer an approximate value for the length of runways for jet aircraft requires knowledge about their function for takeoff and landing of aircraft and requires knowledge of distances required for these operations for specific aircraft. This example appears practical to mechanize. For others, it is more economical to make use of human knowledge and inference.

In order to use this ability of interpreters, facilities for communicating this knowledge are required. Two sorts of communication are provided: object models; and rules for reasoning about models. For object modeling, a high level modeling language has been built, with advanced capabilities. It is described below. For reasoning about models, some of the experience of the Mycin group is being tapped to aid in encoding the knowledge of PI experts. The knowledge is encoded in the form of rules which are used in a backward-chaining system for reasoning about models. The rules are used to predict observables and will be used to suggest matching rules and to make inference from examples. It may be necessary to redesign this reasoning system as practical requirements make increasingly sophisticated demands on the formal basis of the reasoning system. Prediction and matching elements are described below.

5.2.1 The Modeling System

Objects are modeled in a high level language based on a generalized cone representation of primitives in a part/whole graph [Binford]. The representations of most objects are very compact; they are segmented into volume parts which seem natural to the user. This modeling system provides graphic aids for the user for modeling generic objects and relations. The representation also seems natural for machine reasoning because important relationships between surfaces are simply represented in the generalized cone models.

The motivation to build a modeling system can be appreciated by comparison with previous modeling systems. Briefly, they do not provide symbolic results which are necessary for reasoning about generic object classes and generic viewing conditions, and they are not general enough. They can be classified as those based on a few primitives such as cylinders and blocks [Braid] and those based on polyhedra [Baumgart]. Those based on simple primitives were not general enough to represent the objects that were considered. Those based on polyhedra did not have the relationships and the part/whole decomposition needed for reasoning about the models. Previous systems were aimed at hidden surface graphics; in the ACRONYM system, symbolic information about edges and surfaces was needed which was not available in those systems. A previous modeling system based on generalized cones [Miyamoto and Binford] provided a background for the design of the new system.

The object models are embedded in an Object Graph. From the Object Graph, the system predicts an Appearance Graph and an Observability Graph. The relationships of these graphs are shown in figure 1. The Appearance Graph is primarily for the user to visualize the internal models of the system. It is also useful for reasoning about scenes where the viewing conditions are known and specific object models are known. The Appearance Graph is made by computer graphics techniques. Some innovations in graphics were possible because of the generalized cone representation. The predictor and planner has been implemented for a broad class of objects. Several analytic problems were formulated whose solutions have resulted in efficient and general algorithms which are expected to be useful in many tasks. Analytic solutions have been found for symbolic display of generalized cones of two classes: the first class has polygonal cross sections with linear scaling along a circular spine; the second class has circular cross sections with linear scaling along a straight spine. These

analytic solutions are very fast, of order 20 milliseconds. These are two elements of a scheme for a complete analytic solution for prediction for the most interesting subclass of generalized cones. There is reasonable hope that the rest of the analytic solution will be found.

Analytic solutions are crucial for generation of the Observability Graph which requires symbolic relations. They are useful even for the Appearance Graph. In most modeling systems, curved surfaces are displayed by plane faces, with spurious edges, which are suppressed in some systems. In this system, curved surfaces are represented and displayed as such, which contributes greatly to the compactness and to the quality of the display. Consider the analogy with representing curves in two dimensions. To represent a circular arc by straight segments requires many segments for high fidelity, while only a single circular arc is required, or a few segments of a polynomial spline. It is expected that important contributions will be made to *cutting the combinatorics of hidden surface* display because the representations are very compact. That is, in this system, combinatorics are based on the number of objects rather than on the number of approximating planar facets, as in other systems. The symbolic display module is *incomplete*. It does back surface elimination for a fairly general class of objects, but does not yet do complete hidden surface elimination; other capabilities have higher priority. Several designs are being considered.

5.2.2 Prediction: The Observability Graph

The Observability Graph contains generic and specific predictions about shape elements and relations which are observable. When specific objects and viewing conditions are known, predictions about them are included. The heart of the prediction mechanism is the facility for generic predictions. These predictions are in terms of quasi-invariants. These are properties which are stable over a broad range of variants of objects within the model class and stable over a broad range of viewing conditions. They are also selected for visibility, that is easily found by available operators. The predictor which builds the Observability Graph is being implemented in the form of a system of rules in a backward-chaining reasoning system, in consultation with members of the Mycin group. Nodes of the Observability Graph correspond to features which can be obtained from images, that is, surfaces, lines, and "ribbons", which are generalized cones specialized to two dimensions. Since objects are made up of cone primitives, observables for cones and relations between cones are sufficient to generate the complete Observability Graph for all object models.

5.2.3 Matching

The matcher first makes a coarse selection of match candidates based on nodes and arcs of the Observability Graph. It then makes a *detailed match based on the Object Graph*. Each primitive node in an Observability Graph corresponds to a class of ribbons, or a surface; it may be viewed as a predicate which accepts a ribbon or surface with a certain set of attributes. Contextual information provided by related parts or objects of the scene is encoded in arcs between these nodes. The matcher establishes a Linking Graph which has tentative links from nodes of the Observability Graph to those nodes of the Picture Graph which satisfy necessary conditions attached to the nodes of the Observability Graph. It then examines arcs of the Observability Graph to determine consistency between nodes.

Two problems are encountered in matching programs. Errors are made on decisions based on evidence which is too local, and combinatorial search is prohibitive for global decisions. The matcher uses the Observability Graph in several mechanisms for efficiency: first, it uses shape and structure in a powerful

way in obtaining candidates for detailed matching; second, it uses global information in a coarse to fine matching strategy, matching coarse features from the Observability Graph and detailed features from the Object Graph; and third, it will match nodes in order of their cost-effectiveness for achieving the match. The matching will incorporate the structuring of Observability Graphs which was introduced in previous research here at SAIL [Nevatia]. That structuring reduces match combinatorics drastically by imposing a lattice structure on matches. Of these efficiency measures, the effective use of shape prediction and structuring of graph matching are the most powerful. Effective use of shape prediction is a major contribution of this research.

Each match of a subgraph of OG, the Observability Graph, with a subgraph of PG, the Picture Graph, corresponds to an interpretation of that Observability subgraph

interpretation by mapping from Observability Graph to Picture Graph. Typically, there will be multiple spatial relations between edges and ribbons in the Picture Graph, only some of which are consistent with the Observability subgraph. It is, however, a local mapping. The goal then is to determine the best overall interpretation, one which uses the full model. Consider matches between nodes ON of the Observability Graph and nodes PN of the Picture Graph. Global considerations, (particularly structural or spatial relations,) are used to determine whether a pair of ON-PN mappings is consistent. The consistency-finding algorithm now invoked regards each ON-PN correspondence as a node in the "Pairing Graph". Its first task is to use the arcs and relations of the OG to link together consistent pairs of these pairing nodes. It then removes the more isolated nodes from this graph, to leave a large and self-consistent sub-graph.

In the airfield example, the global context primarily involves distinguishing runways from portions of highways among candidate ribbons. Because there are detailed expectations for each interpretation, it is useful to consider each. Locating taxiways, storage areas, and aircraft, nearby large flat areas, and clear flight path along alleged runways supports an airfield interpretation. On the other hand, locating connecting highways, car traffic, buildings and obstructions along the path, supports a highway interpretation.

A simplified version of an Airport serves as an example. Its Object Graph can be briefly described as a collection of several runways and taxiways, close to some terminal and hanger buildings. There will probably be airplanes in the vicinity as well. The system of runways and taxiways should be connected and all these constituent parts of an airport should be in close proximity.

There are both parallel and intersection arcs between runways in the Airport Object Graph. Intersections are usually planar, not overpass intersections. Several runways may be parallel. There will usually be runways in several directions to accomodate wind changes. Further, there is often an underlying equilateral triangle pattern dating back to the time before jets, when runways were much shorter. The glide path will be free of obstructions. Runways are connected by taxiways to terminals or storage areas. A taxiway may be curved, relatively short or hard-to-see.

At the next lower level, these parts must be defined. Runways must be straight, long, level, narrow and visible. In addition, they commonly have markings and a dotted line running down their center, and appear as roads which lead nowhere. (That is, they do not connect into the highway system.) The runway node is itself a graph. with two nodes, the "outline" of the runway, a long straight ribbon with high contrast, and the dotted line down the center of the runway. The range of lengths and widths are approximately known. The sole arc in the runway graph specifies

that the dotted-line ribbon must be contained in the main ribbon and that their axes coincide.

Aircraft are described in terms of graphs whose nodes are volume parts (fuselage, wings, tail, engines) and whose primitives are generalized cones.

There are two types of nodes in the Airport Observability Graph, runways and aircraft. From almost any angle, runways appear as long, straight ribbons with constant width. They usually have markings and boundaries with high contrast. Thus their boundaries or markings are likely to be found by edge finding routines. Runways are more easily found than aircraft for this reason, as well as their length and simple shape. Thus, strategies derived from the Observability Graph are expected to focus attention on runways.

In typical examples, there will be accurate observer altitude, location and orientation and ground elevation. This will enable good approximate estimates for length and width to be made directly from the image. Under these circumstances, typical length and width are observables. In many cases, the images could be registered with familiar observables. For example, in photos of the San Francisco Bay Area, the shore can be registered, to provide a measurement scale over the whole image. Even in other situations, when these quantities could not be included in the Observability Graph, the length to width ratio could be used, as it would be large in almost any viewing situation; and this qualifies it as an observable. In stereo viewing, measurements can be made of flatness and levelness. They would not be observables in monocular viewing. With accurate observer location and information, parallelism is accurately determined. Otherwise, in almost all cases parallelism is nearly preserved. Intersection is invariant. In stereo images, planar intersection can be determined, otherwise it can sometimes be inferred.

Thus far, the edge detection and subsequent ribbon finding process, have been simulated by hand. Also, the interfaces between the Matcher and the Knowledge Base are still being built. The matching process sketched above refers to the driver routines – the real work will be done by the observability functions; that is, the node, arc, and relation predicates.

Past research contributed to these results. The formulation of "generalized translational invariance" provided the generalized cone representation [Binford]. A symbolic display program based on generalized cones preceded this system [Miyamoto]. A program of research led to recognition of a doll, a toy horse, and other complex objects [Nevatia], based on data from a laser ranging system [Agin]. Concepts and algorithms for description of depth maps and for the matching process were demonstrated in that research.

5.3 Progress in Stereo Mapping

New results in stereo mapping contribute to showing the potential of accurate passive ranging. Passive ranging has a survivability advantage over active ranging systems for cruise missiles in hostile environments. Stereo mapping using edge matching has produced stereo maps with a quality adequate for subsequent recognition. Several analytic results have been obtained which will lead to faster and more accurate stereo mapping. The beginnings of a model for stereo mapping systems are beginning to emerge.

Edge-Based Stereo

Results of depth maps of edges were obtained with photos of San Francisco Airport, an apartment building, and a parking lot [Arnold]. The system requires about two minutes of machine time to make a depth map of edges of surfaces. The edge map appears adequate for identification. Edge maps are relatively continuous with few errors. The

depth resolution is sufficiently good that it is possible to tell that the wingtips are higher than the wingroots (dihedral). Some of the weaknesses of current edge operators show up under the close scrutiny of image matching. The system has been rebuilt, with memory management to work with very large images, and is now being tested.

This research aims at high resolution of surface boundaries to make measurement of dimensions and angles. Typically, edge-based techniques offer a factor of 10 improvement in accuracy of determination of surface boundaries over area correlation methods. In correlation, accuracy near a boundary is limited to a fraction of the width of the correlation window (typically 8x8). The Hueckel edge operator, however, provides measurements to a fraction of a pixel, even for weak or noisy edges. Edge-based systems also have an advantage with small objects. Poles and other long, thin objects are prominent features, but are too small for correlation windows.

A serious deficiency of area correlation is failure at surface discontinuities. Simple area correlation techniques inherently fail in the vicinity of surface discontinuities because the edge of an object appears against a different background area in each view of the stereo pair. It is important to locate surface discontinuities, since it is precisely the boundaries of objects where accurate measurements are most important. However, edge operators are ineffective in the presence of texture and most edge operators have problems with smooth shading. In those cases, edge-based techniques encounter problems, while correlation is effective. Thus, edge-based and area-correlation approaches are complementary. The edge-based system will work well in scenes of man-made objects and poorly in natural scenes. For area correlation, the situation is just the opposite. The reason is that man-made objects (cars, buildings) tend to have planar surfaces of uniform intensity and well defined linear edges. Natural surfaces (clouds, trees, hills), on the other hand, are often curved with strong texture and indistinct or irregular boundaries. A general purpose vision system would need to employ both types of techniques.

An essential part of the research is the use of context in matching. The system currently uses local context of edge continuity, and the context of the ground plane. The system is being extended to use context of locally planar surfaces, with successive approximation modeling.

If necessary, a model of the transform from one image to the other can be obtained automatically from the two images, with no knowledge of guidance parameters; an estimate of velocity and time between pictures is useful to estimate the baseline between pictures [Gennery]. Imagine an aircraft approaching a runway. As it moves, objects on both sides appear to move radially outward from a center, the fixed point. The center is the instantaneous direction of motion. The pilot knows that the point which appears stationary is where he will touch down, unless he changes direction. If guidance information about the two images is available, that can be used to eliminate the process of obtaining the camera model. An Interest Operator [Moravec] is applied to the left view to select approximately 50 "interesting" points. A point is "interesting" if it promises to be easily locatable in two dimensions (ie. corners and intersections). A fast binary search correlator [Moravec] produces an initial match for each point, searching the entire right image. The correlator uses a binary search strategy to match points efficiently.

These matches are refined with a high resolution area correlator [Gennery] and passed to a camera model solver [Gennery]. This camera model program solves for four parameters: 1) direction of the stereo axis; 2) relative rotation between left and right views; 3) scale factor between left and right views; 4) in the picture. It is useful to make the

The usual camera solver determines 5 parameters. This form is useful in the degenerate case in which scene heights are small with respect to distance from the camera. The relative positions (disparities) of each matched pair along the stereo axis provide information on heights relative to the film plane. At this stage, about half the original 50 points have been automatically rejected for various reasons. The points and their heights are given to a ground plane finder [Gennery] which fits a plane to a subset of them such that few points are assigned heights below the plane, some may be above the plane, and as many as possible lie on the plane. Total processing for camera model and ground plane is about 8 seconds.

The camera transform provides the information necessary to measure distance of corresponding points. It also determines the stereo axis. Search for matches can then be restricted to one-dimensional searches along the stereo axis, with a great saving in computation.

In any stereo system, ambiguity is a major problem. Edge elements in one view may match with multiple edge elements in the other view. For example, in the parking lot scenes, edges of cars, pavement markings and shadow edges are all parallel and are easily confused. Direction, brightness, colors, and contrast measurements extracted by the edge operator can guide the matching but are not strong conditions. If an edge has continuity in three dimensions, then adjacent, matching edgels along that edge should be continuous in both direction and disparity. Edge continuity and consistency are strong conditions that significantly affect ambiguity. The context of the ground surface is also important in this matching process. A priori constraints may be used during matching to limit the disparity range to that of objects above the ground within a reasonable height.

The next step is to raster-scan the Hueckel edge operator over the two pictures; x-y position, angle of edge, and brightness measures of edge elements are retained. About 1200 edge elements (edgels) are produced from a 128x128 picture in about 18 seconds. Only edge information is used in subsequent stages. Edges from the left and right pictures are transformed into standard coordinate system with the stereo axis in the x direction and disparity shifts due to the tilt of the ground plane cancelled. Thus all points lying on the ground plane will have identical x-y coordinates in the two views.

Edges in the left image are matched with all candidates in the right image. A grid of 8x8 cells is set up for the left and right pictures, each cell being the head of a linked list. Candidates lie in a narrow band vertically, with disparity between the ground plane (zero disparity) and the a priori disparity limit in the x direction. Very loose tests on brightness and angle are made to reject some potential matches. If the match is accepted, a disparity is calculated by projecting the right edgel to the y coordinate of the left edgel. On the average, this search produces 8 ambiguous matches for each edgel, that is, 8 edgels that agree in position, angle and brightness. Most of these ambiguous matches are actually multiple edgels from different positions along the same scene edge.

Edge elements in the left image are linked if continuous in x and y, if their angles match within 90 degrees and they are colinear, and if brightnesses are consistent on at least one side of the edgels. Typically, this produces 3 or 4 links per edgel, and linked edgels tend to follow edges of low to moderate curvature. Time for the matching and linking is 33 seconds.

Each edgel in the left picture has a list of linked edgels and a list of possible matches. An ad hoc "voting" scheme was implemented to establish a consensus for disparities of possible matches of the edgel and those of linked neighbors. Let E be an edgel and L an edgel linked to E. Let dL be a disparity on

L's disparity list and dE a disparity on E's disparity list. If dL and dE are equal or nearly equal (within .125 pixel disparity) then dE gets two votes. If dL and dE are close (within .375 pixel disparity) then dE gets 1 vote. Otherwise, there are no votes. A bell-shaped distribution usually results about the best disparity, with wild or inconsistent matches out on the tails of the curve. The maximum of the distribution is taken as the disparity for E. This processing takes 8 seconds. A file of edgels with their three dimensional locations results.

Results of stereo edge maps for a passenger terminal at San Francisco airport are shown in Figure 2. The matches are sufficiently accurate that the dihedral angle of the wings is noticeable. A small percentage of errors are noticeable in areas with repetitive patterns. The matching is expected to be improved significantly by incorporating the context of surfaces.

Results of stereo edge maps are now being used for aircraft recognition in the ACRONYM system. They have been used with routines aimed toward recognition of vehicles. A rectangle finder has been used to describe the outline of cars in aerial photos. Measurement of the length of a vehicle was accurate to 5%.

The method outlined above suffers from some serious problems. It relies heavily on the Hueckel edge operator. While it may be one of the best available, it is weak at slow intensity gradients, where it finds a multitude of parallel edges that tend to match at every possible disparity. Second, it is a least squares process, and so is easily led astray near corners and in textured areas.

Work on stereo region-growing [Hannah] and motion sequences [Nevatia B] preceded this research. A curve matching program matched parts of images by a coarse-to-fine strategy of matching long curves first [Bolles]. That approach has potential for cruise missile navigation and is being pursued further. The amount of information required by these curve features is quite small, permitting flexible flight paths with reasonable memory requirements.

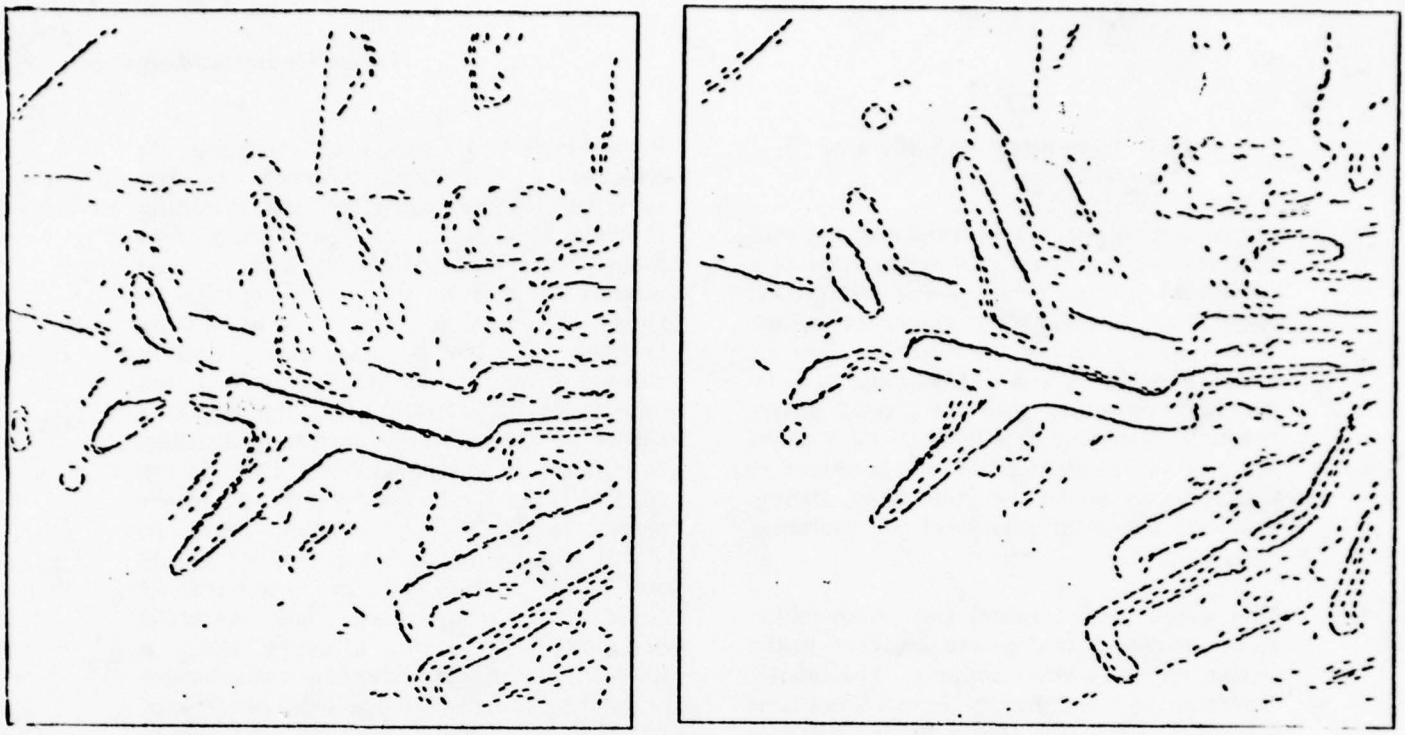


Figure 2a,b. Edge elements from a stereo pair at San Francisco Airport.

Figure 2c. Matched edges near the ground.

Figure 2d. Matched edges above the ground.

5.3.1 Stereo mapping using area correlation

Stereo mapping with area correlation has been extended to make complete stereo maps of a variety of images, aerial photos and ground level, with a completely automated system. The system estimates measurement accuracy for each point. The stereo mapping program has been combined with the ground surface finder to determine objects which stand above the ground. A plane finder has been used to locate planes above the ground, e.g. finding the roof in aerial photos of an apartment house.

The stereo camera model and determination of the stereo calibration are described in the section on stereo edge mapping. The Interest Operator and the Binary Search Correlator are used. Once the stereo camera model is known, the search for matching points is constrained. A match for a given point must lie along a ray in space which projects as a straight line in the other view. Because images are primarily composed of regions corresponding to extended surfaces, areas have matches at approximately the same stereo disparity as neighboring points. For many areas, the search can be eliminated by using this neighborhood context.

The program divides one picture of the pair into square areas, typically 8x8. It selects a starting area and proceeds column by column through the picture. It tries areas adjacent to areas already matched and searches for matches with disparity approximately the same as its neighbors. Previous work here at SAIL [Hannah] used a region growing process. The current approach has an advantage in that with some changes the process requires only portions of the image to be in memory at any one time. That sort of locality is not possible with a region oriented approach.

A high resolution correlator was developed which has several advantages for this purpose. It has increased accuracy of matching. It estimates a probability estimate of the correctness of the match. In the matching sequence, matches are accepted based on the probability estimate. Even in areas of low information content, the noise suppression ability often allows useful results to be obtained. The correlator also produces a measure of the precision of the match. If the information content is too low, the match is not well localized, and the correlator estimates large values for the standard deviations of the position. The standard deviations of match positions are the basis for estimating errors in spatial position measurements. When errors are large, searching can be terminated if desired. In many cases, one standard deviation is large (for example, along a straight edge) but an accurate measurement can still be made unless the eigenvector with large standard deviation lies almost parallel to the stereo axis.

The ground surface finder is used to estimate the ground level. Figure 3 shows a sequence of steps in the operation of the system on a pair of pictures taken from ground level. The final picture shows the heights of image areas which are more than two feet above the ground. Heights are shown by arrows which extend downward from the selected point to the ground directly beneath. It is apparent that the routine succeeds in isolating objects from the ground.

Figure 3a

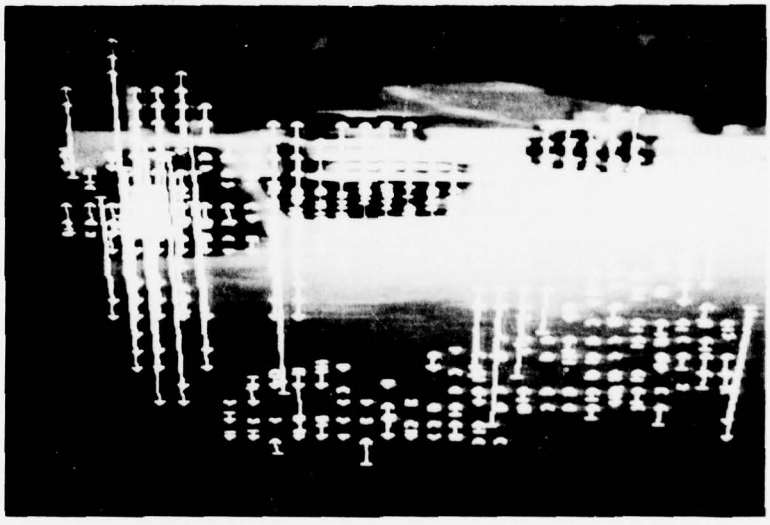


Figure 3b

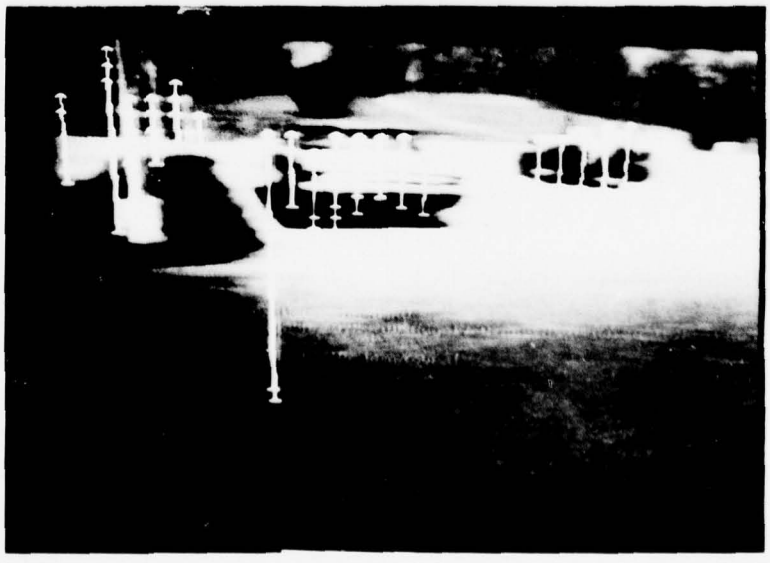


Figure 3b

5.4 References

[Agin] G.J. Agin and T.O. Binford; "Representation and Description of Curved Objects"; IEEE Transactions on Computers; Vol C-25, 440, April 1976;

[Arnold] R.D. Arnold; "Local Context in Matching Edges for Stereo Vision"; Proceedings Image Understanding Workshop, Boston, May 1978.

[Barrow] H.G.Barrow, M.A.Fischler; "An Expert System for Detecting and Interpreting Road Events Depicted in Aerial Imagery"; Proceedings Image Understanding Workshop, Boston, May 1978.

[Baumgart] Baumgart,B.; "Geometric Modeling for Computer Vision"; Stanford AI Lab Memo AIM-249, CS-463, ADA002261, 1974.

[Binford] T.O.Binford; "Visual Perception by Computers"; Invited Paper IEEE Systems Science and Cybernetics; Miami Fla; Dec 1971.

[Bolles] R.C.Bolles; "Verification Vision"; translation perpendicular to the stereo axis.

[Braid] Braid,I.C.; "Designing with Volumes"; Univ of Cambridge, Cantab Press, Cambridge, England, 1973. Braid,I.C.; "Six Systems for Shape Design and Representation - A review"; C.A.D. Group Document No. 87, University of Cambridge.

[Brooks] R. Brooks, R. Greiner, and T.O. Binford; "A Model-Based Vision System"; Proceedings Image Understanding Workshop", Boston, May 1978.

[Gennery] D.B. Gennery; "A Stereo Vision System for Autonomous Vehicles"; Proceedings Image Understanding Workshop, Palo Alto, Oct 1977.

[Hannah] M.J.Hannah; "Computer Matching of Areas in Stereo Images" Stanford University AI Lab Memo AIM-239, July 1974.

[Hueckel] M.H. Hueckel, "A Local Visual Operator which Recognizes Edges and Lines", J. ACM, October 1973.

[Marr], D. Marr; "Analysis of Occluding Contour"; MIT AI Memo 372; October 1976;

[McGhee] R. McGee; "Automatic Recognition of Complex Three-Dimensional Objects from Optical Images"; Ohio State University, 1971; See also: T. Wallace, P.A. Wintz; "3-Dimensional Aircraft Recognition Using Fourier Descriptors"; Proceedings of Image Understanding Workshop, Palo Alto, October 1977.

[Miyamoto] E.Miyamoto and T.O.Binford; "Display Generated by a Generalized Cone Representation"; Conf on Computer Graphics and Image Processing, Anaheim, Cal, May 1975.

[Moravec] H.P. Moravec; "Towards Automatic Visual Obstacle Avoidance" Proceedings 5th International Joint Conf on Artificial Intelligence, MIT, Boston, Aug 1977.

[Nevatia] R. Nevatia and T.O. Binford; "Structured Descriptions of Complex Objects"; Artificial Intelligence 1977.

[Nevatia B] R. Nevatia; Image Processing and Computer Graphics, 1976.

[Shortliffe], E.H. Shortliffe, R. Davis, S.G. Axline, B.G. Buchanan, C.C. Green, and S.N. Cohen; "Computer-Based Consultations in Clinical Therapeutics: Explanation and Rule Acquisition Capabilities of the MYCIN system."; Computers and Biomedical Research, Volume 8, June 1975;

6. Knowledge Based Programming

Personnel: Cordell Green, Jerrold Ginsparg,
Student Research Assistants:
Philippe Cadiou, Richard Gabriel,
Elaine Kant, Juan Ludlow,
Brian McCune, Jorge Philips,
Thomas Pressburger, Louis Steinberg,
Steve Tappel, Stephen Westfold

This section summarizes progress made on the PSI program synthesis system during the past two years. [Green-76B] is an overview of prior work.

A summary of the scope and design of PSI is given, followed by a discussion of its present capabilities. Then a number of examples of PSI in operation are given. Finally publications by the Knowledge Based Programming Group are listed.

6.1 Summary of the PSI Program Synthesis System

The PSI program synthesis system is a computer program that acquires high level descriptions of programs and produces efficient implementations of these programs. Simple symbolic computation programs are specified through dialogues that include natural language, input-output pairs, and partial traces. The programs produced are in LISP, but experiments have shown that the system can be extended to produce code in a block structured language such as PASCAL.

PSI is organized as a collection of interacting modules or programmed experts. The overall design is a group effort, with one individual having responsibility for each module as follows: parser/interpreter, Jerrold Ginsparg; trace and example inference expert, Jorge Phillips; dialogue moderator, Louis Steinberg; explainer, Richard Gabriel; domain expert, Jorge Phillips; program model builder, Brian McCune; coder, Juan Ludlow; and efficiency expert, Elaine Kant. The block diagram in Figure 1 shows these modules. Additional personnel have been working on various projects within these experts. Steve Tappel wrote the rule expander for the program model builder; Stephen Westfold enhanced the examples component of the inference expert; Philippe Cadiou worked on rules for coding procedures; and Thomas Pressburger wrote programs to generate understandable program models and complete handwritten program models.

Personnel previously with the PSI Group made many important contributions. David Barstow developed the coder; Ronny van den Heuvel worked on explication of knowledge about concept formation for the domain expert; Bruce Nelson wrote the program model interpreter; Richard Pattis wrote a general input parser for this interpreter; and Avra Cohn laid a groundwork of domain expertise and general programming knowledge.

The major data paths and modules of the PSI system are shown in Figure 2. There is one data path for each specification method. Currently these are English, input-output examples, and partial traces. A more conventional method, that of a very high level language, is a planned addition to PSI as shown in the diagram. These specifications are integrated in the program net and model.

PSI's operation may be conveniently factored into two parts: the *acquisition phase* (those modules shown above the program model), which acquires the model, and the *synthesis phase* (those modules shown below the program model), which produces a program from the model.

Figure 1: Block Diagram of the PSI Program Synthesis System

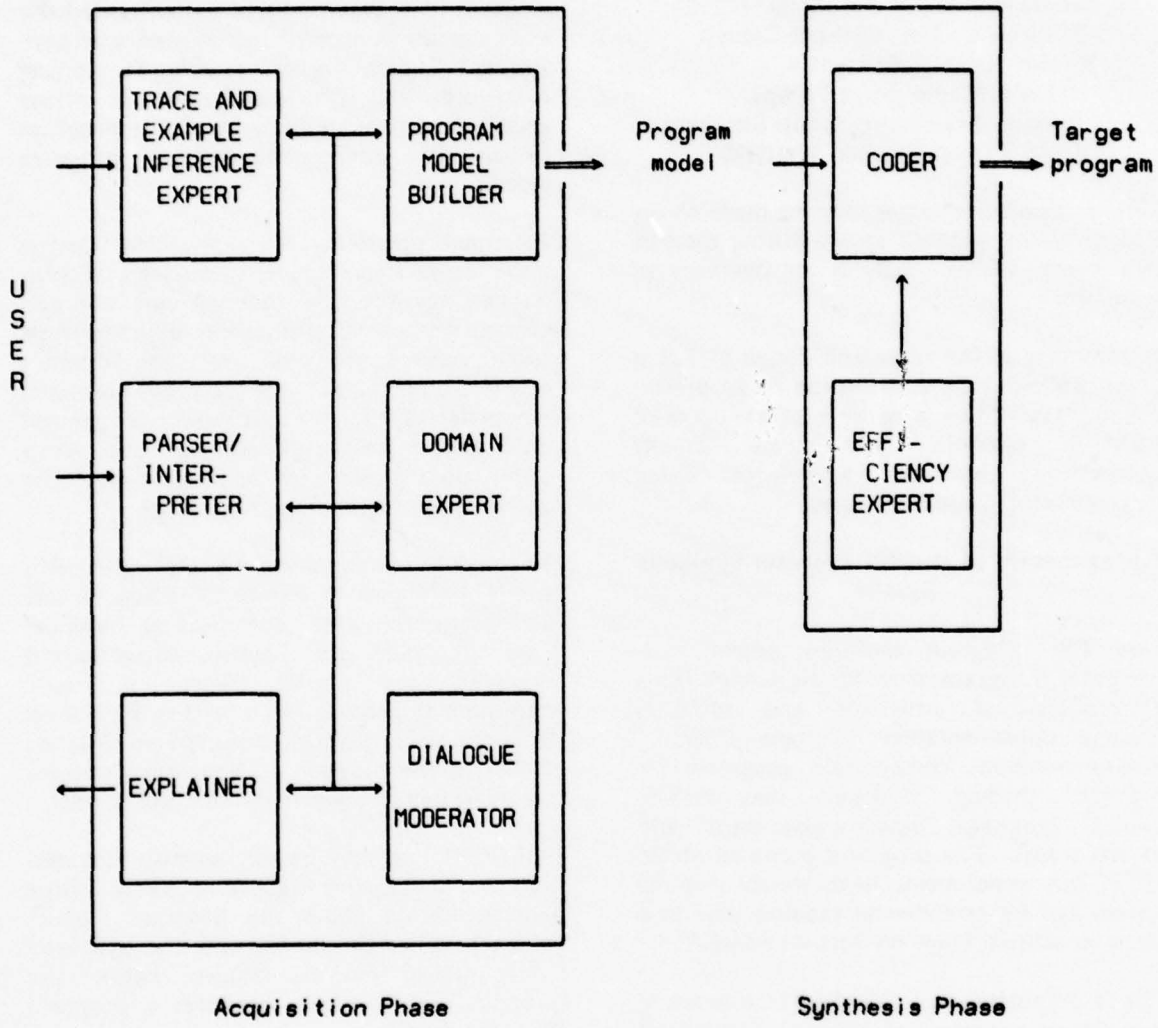
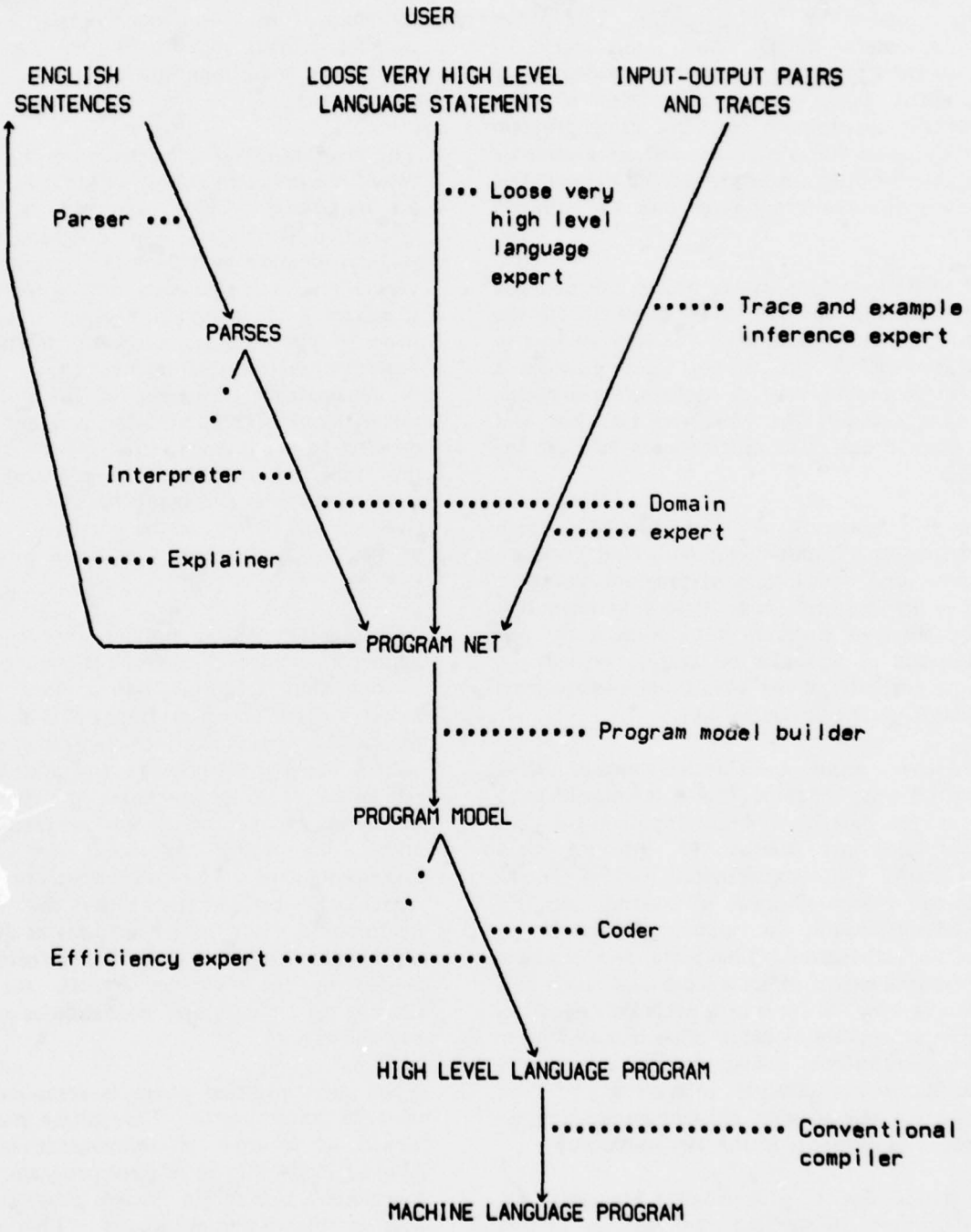


Figure 2: Major Paths of Information Flow in PSI



In the acquisition phase, sentences are first parsed, then interpreted and stored in the program net (also referred to as the "program specification" in [Ginsparg-78]). The parser is a general parser which limits search by incorporating considerable knowledge of English usage. The interpreter is more specific to program synthesis, using program description knowledge as well as knowledge about the question asked and the current topic to facilitate interpretation into the program net.

The dialogue moderator guides the dialogue by selecting or suppressing questions for the user. It attempts to keep PSI and the user in agreement on the current topic, provides a review and preview of topics when the topic changes, helps the user who gets lost, and allows initiative to shift between PSI and the user.

A new module is the explainer, which generates in English reasonably clear questions about and descriptions of program models as they are acquired, in order to help verify that the inferred program description is the one desired. It will also be able to explain the how and why of the acquisition and synthesis process to the interested user.

Another input specification method is a partial trace [Phillips-77]. A trace includes as a special case an example input-output pair. Examples are useful for inferring data structures and simple spatial transformations. Partial traces of states of internal and I/O variables allow the inductive inference of control structures. The trace and example inference expert infers a loose description of a program in the form of a program net, rather than a program model or other true algorithm. This technique allows domain support to disambiguate possible inferences and also separates the issue of efficient implementation from the inference of the user's intention.

Various types of programming knowledge are distributed throughout the modules of the acquisition phase. In contrast, knowledge specific to one particular application domain (e.g., knowledge about learning programs) is concentrated in the domain expert, which supplies domain support by communicating with other acquisition modules through the program net.

The program net and the program model are two of the major interfaces within PSI. Both are high level program and data structure description languages. The program model includes complete, consistent, and interpretable very high level algorithm and information structures. The program net, on the other hand, forms a looser program description. Fragments of the program net can be visited in the order of occurrence of the dialogue, rather than in execution order, and allow less detailed, local, and only partial specification of the program. Since these fragments correspond rather closely to what the user says, they ease the burden of the parser/interpreter as well as the trace and example inference module.

The program model builder [McCune-77] applies knowledge of correct program models to convert the fragments into a model. The model builder processes fragments, checking for completeness and correctness, fills in detail, corrects minor inconsistencies, and adds cross-references. It also generalizes the program description, converting it into a form that allows the coder to look for good implementations. The completed program model may be interpreted by the model interpreter to check that it performs as desired by the user and also to gather information needed by the efficiency expert, such as statistics on set sizes and probabilities of the outcome of tests.

After the acquisition phase is complete, the synthesis phase begins. This phase may be viewed as a series of refinements of the program model into an efficient program, or as a heuristic search for an efficient program that satisfies the program model. The coder

[Barstow-77C] has a body of program synthesis rules [Green & Barstow-75, Green & Barstow-77A] which are applied to gradually transform the program model from abstract into more detailed constructs until it is in the target language. The algorithm and data structures are refined interdependently. The coder deals primarily with the notions of set and correspondence operations and can synthesize programs involving sequences, loops, simple input and output, linked lists, arrays, and hash tables.

The refinement tree effectively forms a planning space that proposes only legal, but possibly inefficient, programs. This tree structure is shared by the coder and the efficiency expert [Kant-77]. When the coder proposes more than one refinement or implementation, the efficiency expert reduces the search by estimating the time-space cost product of each proposed refinement. The better path is followed, and there is no backup unless the estimate later proves to be very bad. An additional method to reduce the size of the search space is the factorization of the program into relatively independent parts so that all combinations of implementations are not considered. An analysis for bottlenecks allows the synthesis effort to concentrate on the more critical parts of the program.

In summary, we have formulated a framework for an automatic program synthesis system and have a start on the kinds of programming knowledge that must be embedded therein.

6.2 Recent Progress on and Present Capabilities of PSI

The entire PSI system can now be used by a knowledgeable user. A number of tests of the entire system have been made. PSI successfully produced LISP code which implements the specifications given in English dialogues for four variants of CLASS, a simple pattern classification program (discussed in detail in the section on "Writing CLASS"). In addition, ten other dialogues have been understood by the parser/interpreter. About three versions of each of five other programs have been coded (some, but not all, of these corresponding to dialogues processed by the parser/interpreter).

The basic system design has remained stable, but many new ideas have been incorporated. The individual capabilities of the modules have been improved and made more robust. An explanation system and one domain module have been added. The interfaces between the modules have been smoothed out so that the entire PSI system can be used without manual intervention.

Work on PSI has been described in two Ph.D. theses, six M.S. projects, six conference papers, and numerous other conference presentations. A journal article on codification of programming knowledge has also been accepted. A complete bibliography is listed under "Publications".

The parser/interpreter now understands over seventy programming concepts and has a vocabulary of more than 175 words. Its programming concepts include data structures (e.g., sets, records), primitive operations (e.g., input, membership), control structures (e.g., loops, conditionals, procedures), and more complicated algorithmic ideas (e.g., user-program interchanges, set construction, quantification). The parser/interpreter is capable of understanding most dialogues which lie within the scope of its concepts and vocabulary. User syntax is not an issue because the parser efficiently parses a very large grammar. The system can sometimes determine the meaning of unknown words (e.g., what concept they represent) from the context in which they appear. The dialogues which the system has understood include those specifying many variants of CLASS, several variants of NEWS (a news story retrieval program), TF (a learning program that uses CLASS as a subroutine), and graph reachability.

The dialogue moderator is capable of choosing which question posed by the parser/interpreter to ask. It has mechanisms (not yet interfaced to the rest of PSI) to answer the question, "Where are we?", and most of the mechanism exists to handle a request to change topic. The moderator has handled dialogues for NEWS and variants of CLASS.

The questions which are asked of the user are now quite readable and coherent. Questions use the same terms as the user did in previous sentences of the dialogue. For example, rather than asking for the definition of "A0018", PSI now asks what it means for "a scene to fit a concept". This new question generation system has been used in the dialogues for CLASS, NEWS, TF, and RECIPE (a recipe retrieval program similar to NEWS but easier to understand). It has produced about twenty substantially different sentence types. The current version should be able to handle all foreseeable dialogues with only minor additions. The question generator is being expanded into a more general explainer which will explain PSI's understanding of the program specification given by the user.

PSI will allow programs to be specified by the use of traces and examples. A version of the trace component of the inference expert was completed which handles simple loop and data structure inference such as that needed for the CLASS and TF dialogues (see the section on "Algorithm Inference from a Trace"). The interface with the parser via the program net has been designed. Implementation is complete except for recognition of when the user is giving a trace rather than continuing the dialogue. The examples component has been greatly improved, and an initial version incorporating our subsetting theories has been implemented. This determines (from an example input-output pair for a certain data object) a suitable program transformation that could have carried the object from its initial to its final state.

An initial version of a domain expert for information retrieval has been implemented. Interfaces with the rest of the system are clearly defined, and a common representational base with the parser/interpreter has been completed. This base, called the program net, has been used by the parser for all the dialogues currently done by the system. The program net has also been used in conjunction with the domain expert for the generation of a variant of NEWS.

A second version of the program model builder has been implemented. Its rule base has increased to 350 rules. The new rules incorporate knowledge of correspondences (or mappings) and primitive operations for accessing them, of procedures and procedure invocations, and of type coercion. The model builder also resolves type-token ambiguities and transforms expressions to canonical forms. A number of program models which are variations on CLASS have been built as part of the entire PSI system. Separately the model builder has successfully constructed the more complex model for RECIPE.

The knowledge base of the coding module has grown to about 450 rules. These rules have been used to code a variety of programs involving graph reachability and prime number finding. The sets and correspondences used in these programs can be represented as lists, arrays, Boolean mappings, or property lists. Several versions of CLASS, RECIPE, NEWS, and TF have been coded. Insertion and selection sorts have also been coded. Rules about reusing the space in arrays have been written and used to synthesize in-place selection and insertion sorts (see the section on "Coding an In-Place Insertion Sort"). Some unnecessary variables in the target code are now eliminated by recomputing previously stored results. This can reduce the number of program variables by a factor of two.

The efficiency expert was used with the coder to write five variants of CLASS, to write

RECIPE, and to write a part of TF. In all cases different implementations are selected when different data structure sizes (for example) are assumed. More than one representation for the same data structure can be used in a program. There are now rules that suggest the circumstances under which various representations are plausible or implausible. This greatly reduces the search space from the original space of all legal programs. Space-time cost estimates are used to compare alternative plausible alternatives. Cost estimates are also used to identify the decisions that may have the greatest impact on the global program cost; the decision making resources are allocated accordingly.

A number of simulated dialogues have been gathered, with a member of the PSI Group playing the role of PSI and people not part of the group as users. The question choosing disambiguate possible inferences and also currently being tested by comparing its behavior with the data from these dialogues.

Preliminary designs have been completed for an additional program specification technique. It is a formal system with the flavor of a very high level programming language. The language allows manipulation of abstract algebraic structures such as mappings and sets. The semantic support available through the domain expert will allow the use of domain specific jargon in this language. This language will allow the user to specify quickly and precisely program descriptions that have already been well thought out.

A system has been written which prints concise, understandable versions of program models in a PASCAL-like notation. The internal representation of the model is designed for programming efficiency and is hard for people to understand. Listings in the concise notation are thus extremely valuable for debugging. Any or all of the parts of a model may be printed, and cross-reference tables are available to index the concise listing and the original model. Listings may be generated for online viewing or printed out for use in documents.

The program model interpreter, which executes models interpretively as an alternative to coding them and running the target program, has been brought completely up to date with the changes to the program modelling language. It has correctly interpreted the ten program models available. The interpreter can now handle the general case of an input statement in which the datum to be input may be of any type occurring in a tree of legal types.

A comparison was made of the running times of interpreted program models versus corresponding compiled LISP functions coded by the PSI synthesis phase. The functions coded by PSI ran up to eleven times faster than the interpreted models for very simple programs. We expect that time savings will grow more than linearly with program complexity.

A rule expander for model building rules is complete, making it easier to write new rules for the program model builder. Rule preconditions are written in a concise declarative language; then the rule expander translates the declarative form into the required fetch and test operations, taking into account any ordering constraints which the preconditions may have and avoiding retesting preconditions unnecessarily.

6.3 Examples of PSI

This section illustrates the performance of the PSI system in doing a number of different tasks. The first example shows a complete run of PSI, from the interactive dialogue to the LISP target code produced, along with output from intermediate stages. The second example is a trace which is input to the inference expert and the program information which is inferred. The final example shows a hand-coded program model for an in-place insertion sort and the LISP program produced by the synthesis phase.

Writing CLASS

This section documents the performance of PSI in writing a program called CLASS. A brief description of the program is followed by listings of the four major forms which its specification assumes during its journey through PSI: an English dialogue between the user and PSI, the program net produced by the parser/interpreter, the program model from the model builder, and the final LISP program generated by the coder.

Description of the Program

CLASS is a simple program that includes many programming constructs necessary for more complex programs. The program specified first inputs a sample set of items (e.g., a list of job qualifications). Then the program repeatedly inputs a trial set (e.g., the qualifications of an applicant), tests whether the sample set (required qualifications) is a subset of the trial set (applicant qualifications), and prints "FIT" or "DIDN'T FIT" accordingly. The program is exited if the word "QUIT" is typed instead of a trial set.

Dialogue

The following dialogue was held with PSI to specify CLASS.

PSI: What is the name of the program you want to write?

USER: CLASS.

PSI: Does CLASS have a special program type?

USER: Classification.

PSI: Please describe CLASS.

USER: CLASS first inputs a concept. Then it repeatedly accepts an input scene from the user, classifies it based on whether or not it fits the concept, and outputs this classification to the user.

PSI: What is the structure of the concept?

USER: A concept is a collection of relations. Each relation is just a string.

PSI: What is the exit test of the loop?

USER: Stop when the user types "QUIT" instead of a scene.

PSI: What is the structure of the scene?

USER: The same as the concept.

PSI: Please define what it means for a scene to fit a concept.

USER: A scene fits a concept if all the relations in the concept are in the scene.

PSI: What is the format of the classification output?

USER: Output "FIT" if the scene fit, "DIDN'T FIT" if it didn't.

PSI: Are we finished?

USER: Yes.

Program Net

A program net is produced by the parser/interpreter, based upon its understanding of the dialogue. The following description is a summary of this net, the algorithmic part being printed in an ALGOL-like notation.

```
A2 is either a set whose generic element
   is a string or a string whose value
   is "QUIT".
A1 is a set whose generic element is a
   string.
A4 is the generic element of A1.
A3 is either TRUE or FALSE.
```

```
B1 is a variable bound to A2.
B2 is a variable bound to A1.
B3 is a variable bound to A4.

CLASS
   PRINT("Ready for the CONCEPT")
   A1 ← READ()
LOOP1:
   PRINT("Ready for the SCENE")
   A2 ← READ()
   IF EQUAL(A2,"QUIT") THEN GO_TO EXIT1
   A3 ← FIT(A2,A1)
   CASES:  IF A3 THEN PRINT("FIT")
      ELSE IF NOT(A3) THEN PRINT("DIDN'T
    FIT")
   GO_TO LOOP1
EXIT1:

FIT(B1,B2)
   FOR_ALL B3
    IMPLIES(MEMBER(B3,B2),MEMBER(B3,B1))
```

Program Model

The program model builder uses the program net produced by the parser/interpreter to construct a complete model of the program. From the internal representation of the resulting program model, the understandable model printer produces the readable form shown on the next page. The actual model also includes much cross-referencing information.

Alternate Implementations

The program model is refined into target language code by the coder and efficiency expert. Dividing PSI into two separate phases allows programs to be optimized by taking different runtime environments into account. The program can be specified once and a program model built. Then by giving different size estimates, probabilities, or cost functions, different target language programs can be produced. The programs will of course have the same input-output behavior, but the code will be optimized differently based on the data structure sizes or other such parameters.

Recall that CLASS reads a sample set of items, then repeatedly inputs a trial set and tests whether the sample set is a subset of the trial set. Since the universe of the sets is not known, a subset test using a bit map, which would be very fast, is not possible. So the subset test is implemented as an enumeration through the elements of the sample set, testing each element for membership in the trial set. When the trial set is small, a simple list (the same as the input format) is a good choice of representation for the sets. The resulting program is the second one shown below.

When the trial set is large, however, it may prove more efficient to convert its represention to a hash table format so that the membership test is much faster. PSI must check whether such savings outweigh the cost of the representation conversion. The resulting program for this case is the third program shown below.

Readable form

```
program CLASS ;

  type
    a0032 : set of string ,
    a0053 : alternative of {string = "QUIT" , a0032} ;

  vars
    a0011 , a0014 , a0035 , a0036 : a0032 ,
    a0055 , m0080 : a0053 ,
    m0095 : string = "DIDN'T FIT" ,
    m0092 : string = "FIT" ,
    m0091 : Boolean ,
    m0081 : string = "QUIT" ;

  procedure a0067(a0036 , a0035 : a0032) : Boolean ;
    a0035 ⊆ a0036 ;

  procedure a0065(a0055 : a0053) : Boolean ;
    a0055 = "QUIT" ;

  begin
    a0011 ← input(a0032 , user , "READY FOR CONCEPT" ,
      "Illegal input.  Input again:  ") ;
    until A0051
    repeat
      begin
        m0080 ← input(a0053 , user , "READY" , "Illegal input.  Input again:  ") ;
        if a0065(m0080) then assert_exit_condition(A0051) ;
        a0014 ← m0080 ;
        m0091 ← a0067(a0014 , a0011) ;
        case
          ¬ m0091 : inform_user("DIDN'T FIT") ;
          m0091 :   inform_user("FIT") ;
        endcase
      end
    finally
      A0051 :
    endloop
  end ;
```

Here is CLASS using a list representation.

```
(CLASS
  [LAMBDA NIL
    (PROG (USER/M0025/C0025)
          (SETQ USER/M0025/C0025 (PROGN (PRIN1 "READY FOR THE CONCEPT")
                                        (TERPRI)
```

algorithm of the dialogue moderator is

```
          (PROG NIL
            RPT/C0059
              [PROG (USER/M0028/C0029 USER/M0021/C0030 USER/M0020/C0032)
                    (SETQ USER/M0020/C0032 (PROGN (PRIN1
                                                    "READY FOR THE SCENE")
                                                  (TERPRI)
                                                  (READ)))
                    (COND
                      ((EQ USER/M0020/C0032 (QUOTE QUIT))
                        (GO L0040)))
                    (SETQ USER/M0021/C0030 USER/M0020/C0032)
                    [SETQ USER/M0028/C0029
                      (NOT (PROG (G0049 G0042)
                                 (SETQ G0049 USER/M0025/C0025)
                           RPT/C0060
                                 (COND
                                   ((NULL G0049)
                                     (GO L0050)))
                                 (SETQ G0042 (CAR G0049))
                                 (COND
                                   ((NOT (MEMBER G0042 USER/M0021/C0030))
                                     (GO L0046)))
                                 (SETQ G0049 (CDR G0049))
                                 (GO RPT/C0060)
                           L0050
                                 (RETURN NIL)
                           L0046
                                 (RETURN T]
                    (COND
                      ((NOT USER/M0028/C0029)
                        (PROGN (PRIN1 "DIDN'T FIT")
                               (TERPRI)))
                      (USER/M0028/C0029 (PROGN (PRIN1 "FIT")
                                               (TERPRI]
              (GO RPT/C0059)
            L0040
              (RETURN])
```

The following version is more efficient for larger sets because the input list is converted to a hash table for processing.

```
(CLASS
  [LAMBDA NIL
    (PROG (USER/M0025/C0025)
          (SETQ USER/M0025/C0025 (PROGN (PRIN1 "READY FOR THE CONCEPT")
                                        (TERPRI)
                                        (READ)))
          (PROG NIL
            RPT/C0065
                [PROG (USER/M0028/C0029 USER/M0021/C0030 USER/M0020/C0032)
                      (SETQ USER/M0020/C0032 (PROGN (PRIN1
                                                      "READY FOR THE SCENE")
                                                    (TERPRI)
                                                    (READ)))
                      (COND
                        ((EQ USER/M0020/C0032 (QUOTE QUIT))
                          (GO L0040)))
                      (SETQ USER/M0021/C0030 (PROG (G0063 G0051 G0052)
                                                   (SETQ G0052 (HARRAY 100))
                                                   (SETQ G0063 USER/M0020/C0032)
                                             RPT/C0067
                                                   (COND
                                                     ((NULL G0063)
                                                       (GO L0064)))
                                                   (SETQ G0051 (CAR G0063))
                                                   (PUTHASH G0051 T G0052)
                                                   (SETQ G0063 (CDR G0063))
                                                   (GO RPT/C0067)
                                             L0064
                                                   (RETURN G0052)))
                      [SETQ USER/M0028/C0029
                        (PROG (USER/T653700/C0036)
                              (SETQ USER/T653700/C0036 USER/M0021/C0030)
                              (RETURN (NOT (PROG (G0049 G0042)
                                                 (SETQ G0049 USER/M0025/C0025)
                                           RPT/C0066
                                                 (COND
                                                   ((NULL G0049)
                                                     (GO L0050)))
                                                 (SETQ G0042 (CAR G0049))
                                                 (COND
                                                   ((NOT (GETHASH G0042
                                                                  USER/T653700/C0036))
                                                     (GO L0046)))
                                                 (SETQ G0049 (CDR G0049))
                                                 (GO RPT/C0066)
                                           L0050
                                                 (RETURN NIL)
                                           L0046
                                                 (RETURN T]
                      (COND
                        ((NOT USER/M0028/C0029)
                          (PROGN (PRIN1 "DIDN'T FIT")
                                 (TERPRI)))
                        (USER/M0028/C0029 (PROGN (PRIN1 "FIT")
                                                 (TERPRI]
                (GO RPT/C0065)
              L0040
                (RETURN])
```

The following is an example run of the CLASS program above.

```
←(CLASS)

READY FOR THE CONCEPT
(MATH ENGLISH BIOLOGY RUSSIAN)

READY FOR THE SCENE
(MATH PHYSICS CHEMISTRY)
DIDN'T FIT

READY FOR THE SCENE
(FRENCH ENGLISH MATH RUSSIAN)
DIDN'T FIT

READY FOR THE SCENE
(BIOLOGY FRENCH RUSSIAN MATH ENGLISH
   CHEMISTRY)
FIT

READY FOR THE SCENE
QUIT
NIL
←
```

Algorithm Inference from a Trace

The trace example in this section shows part of the desired behavior of a program called TF. TF (for "Theory Formation") is a simplified version of Pat Winston's concept formation program. Its goal is to form an internal model of a concept which discriminates between "scenes" which are and are not part of the concept. TF builds up its internal model by repeatedly reading in a scene which may or may not be an instance of the concept. For each scene, TF determines whether it fits the internal model of the concept and verifies this guess with the user. The internal model is then updated based on whether or not the guess was correct. The internal model consists of a set of relations, each marked as being "must" or "may". A scene fits the model if all of the "must" relations are in the instance; "may" relations are optional.

Shown below is a partial trace excerpted from a larger dialogue specifying TF. This excerpt shows the process of updating the concept, given a scene and a fit result.

```
Concept:          ()
Scene:            ((block a)(block b)(on
                    a b))
Result of fit:    True
Updated concept:  (((block a) may)((block
                    b) may)((on a b)
                    may))

Concept:          (((block a) may)((block
                    b) may)((on a b)
                    may))
Scene:            ((block a)(block b))
Result of fit:    False
Updated concept:  (((block a) may)((block
                    b) may)((on a b)
                    must))

Concept:          (((block a) may)((block
                    b) may)((on a b)
                    must))
Scene:            ((block a)(block
                    b)(block c)(on a b))
Result of fit:    True
Updated concept:  (((block a) may)((block
                    b) may)((block c)
                    may)((on a b) must))
```

From this example sequence the inference expert generates the following explanation: If the scene fits the concept, then add all relations in the scene but not present in the concept to the concept and mark them with "may". Otherwise, if the scene doesn't fit the concept, then change the marking of all relations marked "may" in the concept and not appearing in the scene from "may" to "must".

Coding an In-Place Insertion Sort

Programmers commonly use many tricks for saving storage space by reusing space that is no longer needed. The following program illustrates space reutilization in an in-place sort [Green & Barstow-77B]. In this case both the source and target are arrays of size n, and the source may be destroyed. Items are to

be taken one at a time from the source and placed in the target in ascending order. Thus, the source is shrinking at the same rate as the target is growing, and it becomes possible to reduce the total space used from $2n$ to n. The model below for this sort was handwritten.

```
type
  source_set : set of integer destructible ,
  target_set : ordered set of integer ;

vars
  source : source_set ,
  target : target_set reusing source ;

procedure sort(source : source_set) :
    target_set ;
  begin
    target ← target_set{} ;
    ∀ item ∈ source do
      begin
        delete(item , source) ;
        insert(item , target)
      end ;
    return(target)
  end ;
```

Coding rules indicate that the space reduction is possible if both the deletion and addition can be done in place, that is, by always deleting or adding at the same end of the source or target. Other coding rules are then applied to find methods of in-place deletion and insertion. In the resulting program a single array is used, initialized to the source array. The target, initially empty, grows downward from the upper end of the array. The target is kept sorted. Repeatedly the source element at the boundary between source and target is deleted, and a linear scan of the target is used to find where to insert it. All target elements below this point are shifted down one location, and the source element is inserted.

A further improvement of the algorithm, which in this case the coder did not make, is to combine the two scan and shift enumerations into a single enumeration.

Here is one possible implementation, as generated by the coder. Passing the source array as a parameter to the function has been omitted.

```
(SORT
  [LAMBDA NIL
    (PROG (USER_SOURCE_SET_C0118 USER_TARGET_SET_C0120)
          (USER_SOURCE_SET_C0118 ← (<(ARRAY 100) 1 ! 100> )
          (USER_TARGET_SET_C0120 ← ( USER_SOURCE_SET_C0118 :1
                                     USER_SOURCE_SET_C0118 ::2 + 1
                                     ! USER_SOURCE_SET_C0118 ::2 ) )
          (PROG (G0128 G0127 G0124)
                (G0128← USER_SOURCE_SET_C0118)
            RPT_C0130
                (G0127 ← G0128::2)
                (if G0128:2 GT G0128::2  then (GO L0129))
                (G0124 ← (ELT USER_SOURCE_SET_C0118 :1 G0127))
                (PROG (G0132)
                      (G0132←(PROG (G0142)
                                   (G0142 ← USER_TARGET_SET_C0120 :2)
                                RPT_C0144
                                   (if G0142 GT USER_TARGET_SET_C0120 ::2  then (GO L0143))
                                   (if (ELT USER_TARGET_SET_C0120 :1 G0142) GT G0124
                                       then (GO L0143))
                                   (G0142 ← G0142 + 1)
                                   (GO RPT_C0144)
                                L0143
                                                    (READ)))
                      (USER_TARGET_SET_C0120 :2 ← (USER_TARGET_SET_C0120 :2 - 1))
                      (PROG (G0136)
                            (G0136 ← USER_TARGET_SET_C0120 :2 + 1)
                        RPT_C0138
                            (if G0136 GT (G0132 - 1) then (GO L0137))
                            ((ELT USER_TARGET_SET_C0120 :1 (G0136 - 1))
                                  ← (ELT USER_TARGET_SET_C0120 :1 G0136))
                            (G0136←G0136 + 1)
                            (GO RPT_C0138)
                        L0137
                            (RETURN))
                      ((ELT USER_TARGET_SET_C0120 :1 (G0132 - 1))  ← G0124))
                (RPLACD (CDR G0128)
                        ((CDR (CDR G0128)) - 1))
                (GO RPT_C0130)
            L0129
                (RETURN))
          (RETURN USER_TARGET_SET_C0120])
)
```

6.4 Publications

[Barstow-77A] Barstow, David, "A Knowledge Base Organization for Rules about Programming", **Proceedings of the Workshop on Pattern Directed Inference Systems**, *SIGART Newsletter*, Number 63, June 1977, pages 18-22.

[Barstow-77B] Barstow, David R., "A Knowledge Based System for Automatic Program Construction", *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*--1977, Volume 1, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, August 1977, pages 382-388.

[Barstow-77C] Barstow, David R., *Automatic Construction of Algorithms and Data Structures Using a Knowledge Base of Programming Rules*, Ph.D. thesis, Memo AIM-308, Report STAN-CS-77-641, Artificial Intelligence Laboratory, Computer Science Department, Stanford University, Stanford, California, November 1977.

[Barstow & Kant-76] Barstow, David R., and Kant, Elaine, "Observations on the Interaction Between Coding and Efficiency Knowledge in the PSI Program Synthesis System", *Proceedings Second International Conference on Software Engineering*, Computer Society, Institute of Electrical and Electronics Engineers, Inc., Long Beach, California, October 1976, pages 19-31.

[Ginsparg-78] Ginsparg, Jerrold M., *Natural Language Processing in an Automatic Programming Domain*, Ph.D. thesis, Memo AIM-316, Report STAN-CS-78-671, Artificial Intelligence Laboratory, Computer Science Department, Stanford University, Stanford, California, June 1978.

[Green-75] Green, Cordell, "Whither Automatic Programming", invited tutorial lecture, Fourth International Joint Conference on Artificial Intelligence, Tbilisi, USSR, September 1975.

[Green-76A] Green, Cordell, "The PSI Program Synthesis System, 1976", *ACM '76: Proceedings of the Annual Conference*, Association for Computing Machinery, New York, New York, October 1976, pages 74-75.

[Green-76B] Green, Cordell, "The Design of the PSI Program Synthesis System", *Proceedings Second International Conference on Software Engineering*, Computer Society, Institute of Electrical and Electronics Engineers, Inc., Long Beach, California, October 1976, pages 4-18.

[Green-76C] Green, Cordell, "An Informal Talk on Recent Progress in Automatic Programming", *Lectures on Automatic Programming and List Processing*, PIPS-R-12, Electrotechnical Laboratory, Tokyo, Japan, November 1976, pages 1-69.

[Green-77] Green, Cordell, "A Summary of the PSI Program Synthesis System", *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*--1977, Volume 1, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, August 1977, pages 380-381.

[Green-78] Green, Cordell, "The PSI Program Synthesis System, 1978: An Abstract", in Ghosh, Sakti P., and Liu, Leonard Y., editors, *AFIPS Conference Proceedings*: 1978 *National Computer Conference*, Volume 47, AFIPS Press, Montvale, New Jersey, June 1978, pages 673-674.

[Green & Barstow-75] Green, Cordell, and Barstow, David, "Some Rules for the Automatic Synthesis of Programs", *Advance Papers of the Fourth International Joint Conference on Artificial Intelligence*, Volume 1, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, September 1975, pages 232-239.

[Green & Barstow-77A] Green, C. C., and Barstow, D. R., "A Hypothetical Dialogue Exhibiting a Knowledge Base for a Program Understanding System", in Elcock, E. W., and Michie, D., editors, *Machine Intelligence* 8: *Machine Representations of Knowledge*, Ellis Horwood, Ltd., and John Wiley and Sons, Inc., New York, New York, 1977, pages 335-359.

[Green & Barstow-77B] Green, Cordell, and Barstow, David, *On Program Synthesis Knowledge*, Memo AIM-306, Report STAN-CS-77-639, Artificial Intelligence Laboratory, Computer Science Department, Stanford University, Stanford, California, November 1977.

[Green & Barstow-78] Green, Cordell, and Barstow, David, "On Program Synthesis Knowledge", *Artificial Intelligence*, to appear.

[Green et al.-74] Green, C. Cordell, Waldinger, Richard J., Barstow, David R., Elschlager, Robert, Lenat, Douglas B., McCune, Brian P., Shaw, David E., and Steinberg, Louis I., *Progress Report on Program Understanding Systems*, Memo AIM-240, Report STAN-CS-74-444, Artificial Intelligence Laboratory, Computer Science Department, Stanford University, Stanford, California, August 1974.

[Kant-77] Kant, Elaine, "The Selection of Efficient Implementations for a High Level Language", Proceedings of the Symposium on Artificial Intelligence and Programming Languages, *SIGPLAN Notices*, Volume 12, Number 8, *SIGART Newsletter*, Number 64, August 1977, pages 140-146.

[Kant-78] Kant, Elaine, "Efficiency Estimation: Controlling Search in Program Synthesis", in Ghosh, Sakti P., and Liu, Leonard Y., editors, *AFIPS Conference Proceedings*: 1978 *National Computer Conference*, Volume 47, AFIPS Press, Montvale, New Jersey, June 1978, page 703.

[McCune-77] McCune, Brian P., "The PSI Program Model Builder: Synthesis of Very High Level Programs", Proceedings of the Symposium on Artificial Intelligence and Programming Languages, *SIGPLAN Notices*, Volume 12, Number 8, *SIGART Newsletter*, Number 64, August 1977, pages 130-139.

[Phillips-77] Phillips, Jorge V., "Program Inference from Traces Using Multiple Knowledge Sources", *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*—1977, Volume 2, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, August 1977, page 812.

[Shaw et al.-75] Shaw, David E., Swartout, William R., and Green, C. Cordell, "Inferring LISP Programs from Examples", *Advance Papers of the Fourth International Joint Conference on Artificial Intelligence*, Volume 1, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, September 1975, pages 260-267.

Appendix A
Theses

Theses that have been published by this laboratory are listed here. Several earned degrees at institutions other than Stanford, as noted. This list is kept in diskfile THESES [BIB,DOC] @SU-AI.

D. Raj. Reddy, AIM-43
An Approach to Computer Speech Recognition by Direct Analysis of the Speech Wave,
Ph.D. in Computer Science,
September 1966.

S. Persson, AIM-46
Some Sequence Extrapolating Programs: a Study of Representation and Modeling in Inquiring Systems,
Ph.D. in Computer Science, University of California, Berkeley,
September 1966.

Bruce Buchanan, AIM-47
Logics of Scientific Discovery,
Ph.D. in Philosophy, University of California, Berkeley,
December 1966.

James Painter, AIM-44
Semantic Correctness of a Compiler for an Algol-like Language,
Ph.D. in Computer Science,
March 1967.

William Wichman, AIM-56
Use of Optical Feedback in the Computer Control of an Arm,
Eng. in Electrical Engineering,
August 1967.

Monte Callero, AIM-58
An Adaptive Command and Control System Utilizing Heuristic Learning Processes,
Ph.D. in Operations Research,
December 1967.

Donald Kaplan, AIM-60
The Formal Theoretic Analysis of Strong Equivalence for Elemental Properties,
Ph.D. in Computer Science,
July 1968.

Barbara Huberman, AIM-65
A Program to Play Chess End Games,
Ph.D. in Computer Science,
August 1968.

Donald Pieper, AIM-72
The Kinematics of Manipulators under Computer Control,
Ph.D. in Mechanical Engineering,
October 1968.

Donald Waterman, AIM-74
Machine Learning of Heuristics,
Ph.D. in Computer Science,
December 1968.

Roger Schank, AIM-83
(RETURN G0142)))
for a Computer Oriented Semantics,
Ph.D. in Linguistics, University of Texas,
March 1969.

Pierre Vicens, AIM-85
Aspects of Speech Recognition by Computer,
Ph.D. in Computer Science,
March 1969.

Victor D. Scheinman, AIM-92
Design of Computer Controlled Manipulator,
Eng. in Mechanical Engineering,
June 1969.

Claude Cordell Green, AIM-96
The Application of Theorem Proving to Question-answering Systems,
Ph.D. in Electrical Engineering,
August 1969.

James J. Horning, AIM-98
A Study of Grammatical Inference,
Ph.D. in Computer Science,
August 1969.

Michael E. Kahn, AIM-106
The Near-minimum-time Control of Open-loop Articulated Kinematic Chains,
Ph.D. in Mechanical Engineering,
December 1969.

Joseph Becker, AIM-119
An Information-processing Model of Intermediate-Level Cognition,
Ph.D. in Computer Science,
May 1972.

Irwin Sobel, AIM-121
Camera Models and Machine Perception,
Ph.D. in Electrical Engineering,
May 1970.

Michael D. Kelly, AIM-130
Visual Identification of People by Computer,
Ph.D. in Computer Science,
July 1970.

Gilbert Falk, AIM-132
Computer Interpretation of Imperfect Line Data as a Three-dimensional Scene,
Ph.D. in Electrical Engineering,
August 1970.

Jay Martin Tenenbaum, AIM-134
Accommodation in Computer Vision,
Ph.D. in Electrical Engineering,
September 1970.

Lynn H. Quam, AIM-144
Computer Comparison of Pictures,
Ph.D. in Computer Science,
May 1971.

Robert E. Kling, AIM-147
Reasoning by Analogy with Applications to Heuristic Problem Solving: a Case Study,
Ph.D. in Computer Science,
August 1971.

Rodney Albert Schmidt Jr., AIM-149
A Study of the Real-time Control of a Computer-driven Vehicle,
Ph.D. in Electrical Engineering,
August 1971.

Jonathan Leonard Ryder, AIM-155
Heuristic Analysis of Large Trees as Generated in the Game of Go,
Ph.D. in Computer Science,
December 1971.

Jean M. Cadiou, AIM-163
Recursive Definitions of Partial Functions and their Computations,
Ph.D. in Computer Science,
April 1972.

Gerald Jacob Agin, AIM-173
Representation and Description of Curved Objects,
Ph.D. in Computer Science,
October 1972.

Francis Lockwood Morris, AIM-174
Correctness of Translations of Programming Languages – an Algebraic Approach,
Ph.D. in Computer Science,
August 1972.

Richard Paul, AIM-177
Modelling, Trajectory Calculation and Servoing of a Computer Controlled Arm,
Ph.D. in Computer Science,
November 1972.

Aharon Gill, AIM-178
Visual Feedback and Related Problems in Computer Controlled Hand Eye Coordination,
Ph.D. in Electrical Engineering,
October 1972.

Ruzena Bajcsy, AIM-180
Computer Identification of Textured Visiual Scenes,
Ph.D. in Computer Science,
October 1972.

Ashok Chandra, AIM-188
On the Properties and Applications of Programming Schemas,
Ph.D. in Computer Science,
March 1973.

Gunnar Rutger Grape, AIM-201
Model Based (Intermediate Level) Computer Vision,
Ph.D. in Computer Science,
May 1973.

Yoram Yakimovsky, AIM-209
Scene Analysis Using a Semantic Base for Region Growing,
Ph.D. in Computer Science,
July 1973.

Jean E. Vuillemin, AIM-218
Proof Techniques for Recursive Programs,
Ph.D. in Computer Science,
October 1973.

Daniel C. Swinehart, AIM-230
COPILOT: A Multiple Process Approach to Interactive Programming Systems,
Ph.D. in Computer Science,
May 1974.

James Gips, AIM-231
Shape Grammars and their Uses
Ph.D. in Computer Science,
May 1974.

Charles J. Rieger III, AIM-233
Conceptual Memory: A Theory and Computer Program for Processing the Meaning Content of Natural Language Utterances,
Ph.D. in Computer Science,
June 1974.

Christopher K. Riesbeck, AIM-238
Computational Understanding: Analysis of Sentences and Context,
Ph.D. in Computer Science,
June 1974.

Marsha Jo Hannah, AIM-239
Computer Matching of Areas in Stereo Images,
Ph.D. in Computer Science,
July 1974.

James R. Low, AIM-242
Automatic Coding: Choice of Data Structures,
Ph.D. in Computer Science,
August 1974.

Jack Buchanan, AIM-245
A Study in Automatic Programming
Ph.D. in Computer Science,
May 1974.

Neil Goldman, AIM-247
Computer Generation of Natural Language From a Deep Conceptual Base
Ph.D. in Computer Science,
January 1974.

Bruce Baumgart, AIM-249
Geometric Modeling for Computer Vision
Ph.D. in Computer Science,
October 1974.

Ramakant Nevatia, AIM-250
Structured Descriptions of Complex Curved Objects for Recognition and Visual Memory
Ph.D. in Electrical Engineering,
October 1974.

Edward H. Shortliffe, AIM-251
MYCIN: A Rule-Based Computer Program for Advising Physicians Regarding Antimicrobial Therapy Selection
Ph.D. in Medical Information Sciences,
October 1974.

Malcolm C. Newey, AIM-257
Formal Semantics of LISP With Applications to Program Correctness
Ph.D. in Computer Science,
January 1975.

Hanan Samet, AIM-259
Automatically Proving the Correctness of Translations Involving Optimized Coded
PhD in Computer Science,
May 1975.

David Canfield Smith, AIM-260
PYGMALION: A Creative Programming Environment
PhD in Computer Science,
June 1975.

Sundaram Ganapathy, AIM-272
Reconstruction of Scenes Containing Polyhedra From Stereo Pair of Views
Ph.D. in Computer Science,
December 1975.

Linda Gail Hemphill, AIM-273
A Conceptual Approach to Automated Language Understanding and Belief Structures: with Disambiguation of the Word 'For'
Ph.D. in Linguistics,
May 1975.

Norihsa Suzuki, AIM-279
Automatic Verification of Programs with Complex Data Structures
Ph.D. in Computer Science,
February 1976.

Russell Taylor, AIM-282
Synthesis of Manipulator Control Programs From Task-Level Specifications
PhD in Computer Science,
July 1976.

Randall Davis, AIM-283
Applications of Meta Level Knowledge to the Construction, Maintenance and Use of Large Knowledge Bases
Ph.D. in Computer Science,
July 1976.

Rafael Finkel, AIM-284
Constructing and Debugging Manipulator Programs
Ph.D. in Computer Science,
August 1976.

Douglas Lenat, AIM-286
AM: An Artificial Intelligence Approach to Discovery in Mathematics as Heuristic Search
Ph.D. in Computer Science,
July 1976.

Michael Roderick, AIM-287
Discrete Control of a Robot Arm
Engineer in Electrical Engineering,
August 1976.

Robert C. Bolles, AIM-295
Verification Vision Within a Programmable Assembly System
Ph.D. in Computer Science,
December 1976.

Robert Cartwright, AIM-296
Practical Formal Semantic Definition and Verification Systems
Ph.D. in Computer Science,
December 1976.

Todd Wagner, AIM-304
Hardware Verification
PhD in Computer Science,
September 1977.

William Faught, AIM-305
Motivation and Intensionality in a Computer Simulation Model
Ph.D. in Computer Science,
September 1977.

David Barstow, AIM-308
A Conceptual Dependency Representation
Ph.D. in Computer Science,
December 1977.

Appendix B
Film Reports

Prints of the following films are available for distribution. This list is kept in diskfile FILMS [BIB,DOC] @SU-AI.

1. Art Eisenson and Gary Feldman, **Ellis D. Kroptechev and Zeus, his Marvelous Time-sharing System**, 16mm B&W with sound, 15 minutes, March 1967.

The advantages of time-sharing over standard batch processing are revealed through the good offices of the Zeus time-sharing system on a PDP-1 computer. Our hero, Ellis, is saved from a fate worse than death. Recommended for mature audiences only.

2. Gary Feldman, **Butterfinger**, 16mm color with sound, 8 minutes, March 1968.

Describes the state of the hand-eye system at the Artificial Intelligence Project in the fall of 1967. The PDP-6 computer getting visual information from a television camera and controlling an electrical-mechanical arm solves simple tasks involving stacking blocks. The techniques of recognizing the blocks and their positions as well as controlling the arm are briefly presented. Rated "G".

3. Raj Reddy, Dave Espar and Art Eisenson, **Hear Here**, 16mm color with sound, 15 minutes, March 1969.

Describes the state of the speech recognition project as of Spring, 1969. A discussion of the problems of speech recognition is followed by two real time demonstrations of the current system. The first shows the computer learning to recognize phrases and second shows how the hand-eye system may be controlled by voice commands. Commands as complicated as 'Pick up the small block in the lower lefthand corner', are recognized and the tasks are carried out by the computer controlled arm.

4. Gary Feldman and Donald Peiper, **Avoid**, 16mm color, silent, 5 minutes, March 1969.

An illustration of Peiper's Ph.D. thesis. The problem is to move the computer controlled mechanical arm through a space filled with one or more known obstacles. The film shows the arm as it moving through various cluttered environments with fairly good success.

5. Richard Paul and Karl Pingle, **Instant Insanity**, 16mm color, silent, 6 minutes, August, 1971.

Shows the hand/eye system solving the puzzle *Instant Insanity*. Sequences include finding and recognizing cubes, color recognition and object manipulation. [Made to accompany a paper presented at the 1971 IJCAI. May be hard to understand without a narrator.]

6. Suzanne Kandra, **Motion and Vision**, 16mm color, sound, 22 minutes, November 1972.

A technical presentation of three research projects completed in 1972: advanced arm control by R. P. Paul [AIM-177], visual feedback control by A. Gill [AIM-178], and representation and description of curved objects by G. Agin [AIM-173]. Drags a bit.

7. Larry Ward, **Computer Interactive Picture Processing**, (MARS Project), 16mm color, sound, 8 min., Fall 1972.

This film describes an automated picture differencing technique for analyzing the variable surface features on Mars using data returned by the Mariner 9 spacecraft. The system uses a time-shared, terminal oriented PDP-10 computer. The film proceeds at a breathless pace. Don't blink, or you will miss an entire scene.

8. D.I. Okhotsimsky, et al, **Display Simulations of 6-Legged Walking**, Institute of Applied Mathematics – USSR Academy of Science, (titles translated by Stanford AI Lab and edited by Suzanne Kandra), 16mm black and white, silent, 10 minutes, 1972.

A display simulation of a 6-legged ant-like walker getting over various obstacles. The research is aimed at a planetary rover that would get around by walking. This cartoon favorite beats Mickey Mouse hands down. Or rather, feet down.

9. Richard Paul, Karl Pingle, and Bob Bolles, ***Automated Pump Assembly***, 16mm color, silent (runs at sound speed!), 7 minutes, April, 1973.

Shows the hand-eye system assembling a simple pump, using vision to locate the pump body and to check for errors. The parts are assembled and screws inserted, using some special tools designed for the arm. Some titles are included to help explain the film.

10. Terry Winograd, **Dialog with a robot**, MIT A. I. Lab., 16mm black and white, silent, 20 minutes, 1971.

Presents a natural language dialog with a simulated robot block-manipulation system. The dialog is substantially the same as that in *Understanding Natural Language* (T. Winograd, Academic Press, 1972). No explanatory or narrative material is on the film.

11. Karl Pingle, Lou Paul, and Bob Bolles, **Programmable Assembly, Three Short Examples**, 16mm color, sound, 8 minutes, October 1974.

The first segment demonstrates the arm's ability to dynamically adjust for position and orientation changes. The task is to mount a bearing and seal on a crankshaft. Next, the arm is shown changing tools and recovering from a run-time error. Finally, a cinematic first: *two* arms cooperating to assemble a hinge.

12. Brian Harvey, **Display Terminals at Stanford**, 16mm B&W, sound, 13 minutes, May 1975.

Although there are many effective programs to use display terminals for special graphics applications, very few general purpose timesharing systems provide good support for using display terminals in normal text display applications. This film shows a session using the display system at the Stanford AI Lab, explaining how the display support features in the Stanford monitor enhance the user's control over his job and facilitate the writing of display-effective user programs.

Appendix C
External Publications

Articles and books by project members that have appeared since July 1973 are listed here alphabetically by lead author. Earlier publications are given in our ten-year report [Memo AIM-228] and in diskfile PUBS.OLD [BIB,DOC] @SU-AI. The list below is kept in PUBS [BIB,DOC] @SU-AI.

1. Agin, Gerald J., Thomas O. Binford, **Computer Description of Curved Objects**, *Proceedings of the Third International Joint Conference on Artificial Intelligence*, Stanford University, August 1973.

2. Agin, G.J., T.O. Binford; **Representation and Description of Curved Objects**, *IEEE Transactions on Computers*, Vol C-25, 440, April 1976.

3. Aiello, Mario, Richard Weyhrauch, **Checking Proofs in the Metamathematics of First Order Logic**, *Adv. Papers of 4th Int. Joint Conference on Artificial Intelligence*, Vol. 1, pp. 1-8, September 1975.

4. Arnold, R.D., **Local Context in Matching Edges for Stereo Vision**, *Proc. Image Understanding Workshop*, Boston, May 1978.

5. Ashcroft, Edward, Zohar Manna, Amir Pnueli, **Decidable Properties of Monodic Functional Schemas**, *J. ACM*, Vol. 20, No. 3, pp. 489-499, July 1973.

6. Ashcroft, Edward, Zohar Manna, **Translating Program Schemas to While-schemas**, *SIAM Journal on Computing*, Vol. 4, No. 2, pp. 125-146, June 1975.

7. Bajcsy, Ruzena, **Computer Description of Textured Scenes**, *Proc. Third Int. Joint Conf. on Artificial Intelligence*, Stanford U., 1973.

8. Barstow, David, Elaine Kant, **Observations on the Ineraction between Coding and Efficiency Knowledge in the PSI Program Synthesis System**, *Proc. 2nd Int. Conf. on Software Engineering*, IEEE Computer Society, Long Beach, California, October 1976.

9. Barstow, David, **A Knowledge-Based System for Automatic Program Construction**, *Proc. Int. Joint Conf. on A.I.*, August 1977.

10. Biermann, A. W., R.I. Baum, F.E. Petry, **Speeding Up the Synthesis of Programs from Traces**, *IEEE Trans. Computers*, February 1975.

11. Bobrow, Daniel, Terry Winograd, **An Overview of KRL, a Knowledge Representation Language**, *J. Cognitive Science*, Vol. 1, No. 1, 1977.

12. Bobrow, Dan, Terry Winograd, & KRL Research Group, **Experience with KRL-0: One Cycle of a Knowledge Representation Language**, *Proc. Int. Joint Conf. on A.I.*, August 1977.

13. Bolles, Robert C. **Verification Vision for Programmable Assembly**, *Proc. Int. Joint Conf. on A.I.*, August 1977.

14. Brooks, R., R. Greiner, and T.O. Binford, **A Model-Based Vision System**; *Proc. Image Understanding Workshop*, Boston, May 1978.

15. Cartwright, Robert S., Derek C. Oppen, **Unrestricted Procedure Calls in Hoare's Logic**, *Proc. Fifth ACM Symposium on Principles of Programming Languages*, January 1978.

16. Chandra, Ashok, Zohar Manna, **On the Power of Programming Features**, *Computer Languages*, Vol. 1, No. 3, pp. 219-232, September 1975.

17. Chowning, John M., **The Synthesis of Complex Audio Spectra by means of Frequency Modulation,** *J. Audio Engineering Society*, September 1973.

18. Clark, Douglas, and Green, C. Cordell, **An Empirical Study of List Structure in LISP,** *Communications of the ACM*, Volume 20, Number 2, February 1977.

19. Colby, Kenneth M., *Artificial Paranoia: A Computer Simulation of the Paranoid Mode*, Pergamon Press, N.Y., 1974.

20. Colby, K.M. and Parkison, R.C. **Pattern-matching rules for the Recognition of Natural Language Dialogue Expressions,** *American Journal of Computational Linguistics*, 1, September 1974.

21. Dershowitz, Nachum, Zohar Manna, **On Automating Structural Programming,** *Colloques IRIA on Proving and Improving Programs*, Arc-et-Senans, France, pp. 167-193, July 1975.

22. Dershowitz, Nachum, Zohar Manna, **The Evolution of Programs: a system for automatic program modification,** *IEEE Trans. Software Eng.*, Vol. 3, No. 5, pp. 377-385, November 1977.

23. Dershowitz, Nachum, Zohar Manna, **Inference Rules for Program Annotation, Automatic Construction of Algorithms** *Engineering*, Atlanta, Ga., pp. 158-167, May 1978.

24. Dobrotin, Boris M., Victor D. Scheinman, **Design of a Computer Controlled Manipulator for Robot Research,** *Proc. Third Int. Joint Conf. on Artificial Intelligence*, Stanford U., 1973.

25. Enea, Horace, Kenneth Mark Colby, **Idiolectic Language-Analysis for Understanding Doctor-Patient Dialogues,** *Proceedings of the Third International Joint Conference on Artificial Intelligence*, Stanford University, August 1973.

26. Faught, William S., **Affect as Motivation for Cognitive and Conative Processes,** *Adv. Papers of 4th Int. Joint Conference on Artificial Intelligence*, Vol. 2, pp. 893-899, September 1975.

27. Feldman, Jerome A., James R. Low, **Comment on Brent's Scatter Storage Algorithm,** *Comm. ACM*, November 1973.

28. Feldman, Jerome A., Yoram Yakimovsky, **Decision Theory and Artificial Intelligence: I A Semantics-based Region Analyzer,** *Artificial Intelligence J.*, Vol. 5, No. 4, Winter 1974.

29. Finkel, Raphael, Russell Taylor, Robert Bolles, Richard Paul, Jerome Feldman, **An Overview of AL, a Programming System for Automation,** *Adv. Papers of 4th Int. Joint Conference on Artificial Intelligence*, Vol. 2, pp. 758-765, September 1975.

30. Floyd, Robert, Louis Steinberg, **An Adaptive Algorithm for Spatial Greyscale,** *Proc. Society for Information Display*, Volume 17, Number 2, pp. 75-77, Second Quarter 1976.

31. Fuller, Samuel H., Forest Baskett, **An Analysis of Drum Storage Units,** *J. ACM*, Vol. 22, No. 1, January 1975.

32. Funt, Brian, **WHISPER: A Problem-solving System utilizing Diagrams and a Parallel Processing Retina,** *Proc. Int. Joint Conf. on A.I.*, August 1977.

33. Gennery, Don **A Stereo Vision System for an Autonomous Vehicle,** *Proc. Int. Joint Conf. on A.I.*, August 1977.

34. Gennery, D.B., **A Stereo Vision System for Autonomous Vehicles,** *Proc. Image Understanding Workshop*, Palo Alto, Oct 1977.

35. German, Steven, **Automating Proofs of the Absence of Common Runtime Errors**, *Proc. Fifth ACM Symposium on Principles of Programming Languages*, January 1978.

36. Goldman, Neil M., **Sentence Paraphrasing from a Conceptual Base**, *Comm. ACM*, February 1975.

37. Goldman, Ron, **Recent Work with the AL System**, *Proc. Int. Joint Conf. on A.I.*, August 1977.

38. Green, Cordell, David Barstow, **Some Rules for the Automatic Synthesis of Programs**, *Adv. Papers of 4th Int. Joint Conference on Artificial Intelligence*, Vol. 1, pp. 232-239, September 1975.

39. Green, Cordell, and Barstow, David, **Some Rules for the Automatic Synthesis of Programs**, *Advance Papers of the Fourth International Joint Conference on Artificial Intelligence*, Volume 1, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, September 1975, pages 232-239.

40. Green, Cordell, **The Design of the PSI Program Synthesis System**, *Proc. 2nd Int. Conf. on Software Engineering*, IEEE Computer Society, Long Beach, California, October 1976.

41. Green, Cordell, **The PSI Program Synthesis System, 1976**, *ACM '76: Proceedings of the Annual Conference*, Association for Computing Machinery, New York, New York, October 1976, pages 74-75.

42. Green, C. C., and Barstow, D. R., **A Hypothetical Dialogue Exhibiting a Knowledge Base for a Program Understanding System**, in Elcock, E. W., and Michie, D., editors, *Machine Intelligence 8: Machine Representations of Knowledge*, Ellis Horwood, Ltd., and John Wiley and Sons, Inc., New York, New York, 1976.

43. Green, C. C., **A Summary of the PSI Program Synthesis System**, *Proc. Int. Joint Conf. on A.I.*, August 1977.

44. Harvey, Brian, **Increasing Programmer Power at Stanford with Display Terminals**, *Minutes of the DECsystem-10 Spring-75 DECUS Meeting*, Digital Equipment Computer Users Society, Maynard, Mass., 1975.

45. Hieronymus, J. L., N. J. Miller, A. L. Samuel, **The Amanuensis Speech Recognition System**, *Proc. IEEE Symposium on Speech Recognition*, April 1974.

46. Hieronymus, J. L., **Pitch Synchronous Acoustic Segmentation**, *Proc. IEEE Symposium on Speech Recognition*, April 1974.

47. Hilf, Franklin, **Use of Computer Assistance in Enhancing Dialog Based Social Welfare, Public Health, and Educational Services in Developing Countries**, *Proc. 2nd Jerusalem Conf. on Info. Technology*, July 1974.

48. Hilf, Franklin, **Dynamic Content Analysis**, *Archives of General Psychiatry*, January 1975.

49. Hueckel, Manfred H., **A Local Visual Operator which Recognizes Edges and Lines**, *J. ACM*, October 1973.

50. Igarashi, S., R. L. London, D. C. Luckham, **Automatic Program Verification I: Logical Basis and its Implementation**, *Acta Informatica*, , Vol. 4, pp.145-182, March 1975.

51. Ishida, Tatsuzo, **Force Control in Coordination of Two Arms**, *Proc. Int. Joint Conf. on A.I.*, August 1977.

52. Kant, Elaine, **The Selection of Efficient Implementations for a High-level Language**, *Proc. SIGART-SIGPLAN Symp. on A.I. & Prog. Lang.*, August 1977.

53. Karp, Richard A., David C Luckham, **Verification of Fairness in an Implementation of Monitors**, *Proc. 2nd Intnl. Conf. on Software Engineering*, PP. 40-46, October 1976.

54. Katz, Shmuel, Zohar Manna, **A Heuristic Approach to Program Verification**, *Proc. Third Int. Joint Conf. on Artificial Intelligence*, Stanford University, pp. 500-512, August 1973.

55. Katz, Shmuel, Zohar Manna, **Towards Automatic Debugging of Programs**, *Proc. Int. Conf. on Reliable Software*, Los Angeles, April 1975.

56. Katz, Shmuel, Zohar Manna, **Logical Analysis of Programs**, *Comm. ACM*, Vol. 19, No. 4, pp. 188-206, April 1976.

57. Katz, Shmuel, Zohar Manna, **A Closer Look at Termination**, *Acta Informatica*, Vol. 5, pp. 333-352, April 1977.

58. Lenat, Douglas B., **BEINGS: Knowledge as Interacting Experts**, *Adv. Papers of 4th Int. Joint Conference on Artificial Intelligence*, Vol. 1, pp. 126-133, September 1975.

59. Luckham, David C., **Automatic Problem Solving**, *Proceedings of the Third International Joint Conference on Artificial Intelligence*, Stanford University, August 1973.

60. Luckham, David C., Jack R. Buchanan, **Automatic Generation of Programs Containing Conditional Statements**, *Proc. AISB Summer Conference*, U. Sussex, July 1974.

61. Luckham, David C., Nori Suzuki, **Proof of Termination within a Weak Logic of Programs**, *Acta Informatica*, Vol 8, No. 1, pp. 21-36, March 1977.

62. Luckham, David C., **Program Verification and Verification-oriented Programming**, *Proc. I.F.I.P. Congress '77*, August 1977.

63. Manna, Zohar, **Program Schemas**, in *Currents in the Theory of Computing* (A. V. Aho, Ed.), Prentice-Hall, Englewood Cliffs, N. J., 1973.

64. Manna, Zohar, Stephen Ness, Jean Vuillemin, **Inductive Methods for Proving Properties of Programs**, *Comm. ACM*, Vol. 16, No. 8, pp. 491-502, August 1973.

65. Manna, Zohar, **Automatic Programming**, *Proceedings of the Third International Joint Conference on Artificial Intelligence*, Stanford University, August 1973.

66. Manna, Zohar, *Mathematical Theory of Computation*, McGraw-Hill, New York, 1974.

67. Manna, Zohar, Amir Pneuli, **Axiomatic Approach to Total Correctness**, *Acta Informatica*, Vol. 3, pp. 243-263, 1974.

68. Manna, Zohar, Richard Waldinger, **Knowledge and Reasoning in Program Synthesis**, *Artificial Intelligence*, Vol. 6, pp. 175-208, 1975.

69. Manna, Zohar, Adi Shamir, **The Theoretical Aspects of the Optimal Fixpoint**, *SIAM Journal of Computing*, Vol. 5, No. 3, pp.414-426, September 1976.

70. Manna, Zohar, Richard Waldinger, **The Automatic Synthesis of Recursive Programs**, *Proc. SIGART–SIGPLAN Symp. on A.I. & Prog. Lang.*, August 1977.

71. Manna, Zohar, Richard Waldinger, **The Automatic Synthesis of Systems of Recursive Programs**, *Proc. Int. Joint Conf. on A.I.*, August 1977.

72. Manna, Zohar, Adi Shamir, **The Optimal-Fixpoint Approach to Recursive Programs**, *Comm. ACM*, Vol. 20, No. 11, pp. 824-831, November 1977.

73. Manna, Zohar, Richard Waldinger, (eds.), *Studies in Automatic Programming Logic*, American Elsevier, New York, NY, 1977.

74. Manna, Zohar, Richard Waldinger, **Is 'Sometime' sometimes better than 'Always'? Intermittant Assertions in Proving Program Correctness**, *Comm. ACM*, Vol. 21, No. 2, pp. 159-172, February 1978.

75. Manna, Zohar, Adi Shamir, **The Convergence of Functions to Fixpoints of Recursive Definitions**, *Theoretical Conputer Science J.*, Vol. 6, pp. 109-141, March 1978.

76. Manna, Zohar, Richard Waldinger, **The Logic of Computer Programming**, *IEEE Trans. Software Eng.*, Vol. SE-4, No. 5, pp. 199-224, May 1978.

77. Manna, Zohar, Richard Waldinger, **The Synthesis of Structure-changing Programs**, *Proc. 3rd Int. Conf. on Software Eng.*, Atlanta, GA, May 1978.

78. Manna, Zohar, Richard Waldinger, **The DEDALUS System**, *Proc. National Computer Conf.*, Anaheim, CA, June 1978.

79. McCarthy, John, **Mechanical Servants for Mankind**, *Britannica Yearbook of Science and the Future*, 1973.

Proc. 3rd Int. Conf. on Software **Intelligence: A General Survey** by Sir James Lighthill, *Artificial Intelligence*, Vol. 5, No. 3, Fall 1974.

81. McCarthy, John, **Modeling Our Minds** *Science Year* 1975, The World Book Science Annual, Field Enterprises Educational Corporation, Chicago, 1974.

82. McCarthy, John, **Proposed Criterion for a Cipher to be Probable-word-proof**, *Comm. ACM*, February 1975.

83. McCarthy, John, **An Unreasonable Book**, a review of *Computer Power and Human Reason* by Joseph Weizenbaum (W.H. Freeman and Co., San Francisco, 1976), *SIGART Newsletter* #58, June 1976.

84. McCarthy, John, Review: *Computer Power and Human Reason*, by Joseph Weizenbaum (W.H. Freeman and Co., San Francisco, 1976) in *Physics Today*, 1977.

85. McCarthy, John, **Another SAMEFRINGE**, *SIGART Newsletter* No. 61, February 1977.

86. McCarthy, John, **The Home Information Terminal**, *The Grolier Encyclopedia*, 1977.

87. McCarthy, John, M. Sato, T. Hayashi, S. Igarashi, **On the Model Theory of Knowledge**, *Proc. Int. Joint Conf. on A.I.*, August 1977.

88. McCarthy, John, **Epistemological Problems of Artificial Intelligence**, *Proc. Int. Joint Conf. on A.I.*, August 1977.

89. McCarthy, John, **History of LISP**, *Proc. ACM Conf. on History of Programming Languages*, 1978.

90. McCarthy, John, **Representation of Recursive Programs in First Order Logic,** *Proc. Int. Conf. on Mathematical Studies of Information Processing*, Kyoto Japan, 1978.

91. McCarthy, John, **Ascribing Mental Qualities to Machines,** *Philosophical Perspectives in Artificial Intelligence*, Martin Ringle (ed.), Humanities Press, to appear 1978.

92. McCune, Brian, **The PSI Program Model Builder: Synthesis of Very High-level Programs,** *Proc. SIGART–SIGPLAN Symp. on A.I. & Prog. Lang.*, August 1977.

93. Miller, N. J., **Pitch Detection by Data Reduction,** *Proc. IEEE Symposium on Speech Recognition*, April 1974.

94. Moore, Robert C., **Reasoning about Knowledge and Action,** *Proc. Int. Joint Conf. on A.I.*, August 1977.

95. Moorer, James A., **The Optimum Comb Method of Pitch Period Analysis of Continuous Speech,** *IEEE Trans. Acoustics, Speech, and Signal Processing*, Vol. ASSP-22, No. 5, October 1974.

96. Moorer, James A., **On the Transcription of Musical Sound by Computer,** *USA–JAPAN Computer Conference*, August 1975.

97. Morales, Jorge J., **Interactive Theorem Proving,** *Proc. ACM National Conference*, August 1973.

98. Moravec, Hans, **Towards Automatic Visual Obstacle Avoidance,** *Proc. Int. Joint Conf. on A.I.*, August 1977.

99. Moravec, Hans, **A Non-synchronous Orbital Skyhook,** *J. Astronautical Sciences*, Vol 26, No. 1, 1978.

100. Nelson, C. G., Oppen, D. C., **Fast Decision Algorithms based on UNION and FIND,** *Proc. 18th Annual IEEE Symposium on Foundations of Computer Science*, October 1977.

101. Nelson, C. G., Derek Oppen, **A Simplifier based on Fast Decision Algorithms,** *Proc. Fifth ACM Symposium on Principles of Programming Languages*, January 1978.

102. Nevatia, Ramakant, Thomas O. Binford, **Structured Descriptions of Complex Objects,** *Proceedings of the Third International Joint Conference on Artificial Intelligence*, Stanford University, August 1973.

103. Nevatia, R., T.O. Binford; **Structured Descriptions of Complex Objects;** *Artificial Intelligence*, 1977.

104. Newell, A., Cooper, F. S., Forgie, J. W., Green, C. C., Klatt, D. H., Medress, M. F., Neuburg, E. P., O'Malley, M. H., Reddy, D. R., Ritea, B., Shoup, J. E., Walker, D. E., and Woods, W. A., *Considerations for a Follow-On ARPA Research Program for Speech Understanding Systems*, Information Processing Techniques Office, Advanced Research Projects Agency, Department of Defense, Arlington, Virginia, August 1975.

105. Oppen, Derek, S.A. Cook, **Proving Assertions about Programs that Manipulate Data Structures,** *Acta Informatica*, Vol. 4, No. 2, pp. 127-144, 1975.

106. Oppen, Derek C., **Reasoning about Recursive Data Structures,** *Proc. Fifth ACM Symposium on Principles of Programming Languages*, January 1978.

107. Oppen, Derek C., **A Superexponential Bound on the Complexity of Presburger Arithmetic,** *Journal of Computer and Systems Sciences*, June 1978.

108. Phillips, Jorge, T. H. Bredt, **Design and Verification of Real-time Systems**, *Proc. 2nd Int. Conf. on Software Engineering*, IEEE Computer Society, Long Beach, California, October 1976.

109. Phillips, Jorge, **Program Inference from Traces using Multiple Knowledge Sources**, *Proc. Int. Joint Conf. on A.I.*, August 1977.

110. Quam, Lynn, Robert Tucker, Botond Eross, J. Veverka and Carl Sagan, **Mariner 9 Picture Differencing at Stanford**, *Sky and Telescope*, August 1973.

111. Rubin, Jeff, **Computer Communication via the Dial-up Network**, *Minutes of the DECsystem-10 Spring-75 DECUS Meeting*, Digital Equipment Computer Users Society, Maynard, Mass., 1975.

112. Sagan, Carl, J. Veverka, P. Fox, R. Dubisch, R. French, P. Gierasch, L. Quam, J. Lederberg, E. Levinthal, R. Tucker, B. Eross, J. Pollack, **Variable Features on Mars II: Mariner 9 Global Results**, *J. Geophys. Res.*, 78, 4163-4196, 1973.

113. Samet, Hanan, **Proving the Correctness of Heuristically Optimized Code**, *Comm. ACM*, July 1978.

114. Schank, Roger C., Neil Goldman, Charles J. Rieger III, Chris Riesbeck, **MARGIE: Memory, Analysis, Response Generation and Inference on English**, *Proceedings of the Third International Joint Conference on Artificial Intelligence*, Stanford University, August 1973.

115. Schank, Roger C., Kenneth Colby (eds), *Computer Models of Thought and Language*, W. H. Freeman, San Francisco, 1973.

116. Schank, Roger, **The Conceptual Analysis of Natural Language**, in R. Rustin (ed.), *Natural Language Processing*, Algorithmics Press, New York, 1973.

117. Schank, Roger, Charles J. Rieger III, **Inference and Computer Understanding of Natural Language**, *Artificial Intelligence J.*, Vol.5, No. 4, Winter 1974.

118. Schank, Roger C., Neil M. Goldman, Charles J. Rieger III, Christopher K. Riesbeck, **Interface and Paraphrase by Computer**, *J. ACM*, Vol 22, No. 3, July 1975.

119. Shaw, David E., William R. Swartout, C. Cordell Green, **Inferring LISP Programs from Examples**, *Adv. Papers of 4th Int. Joint Conference on Artificial Intelligence*, Vol. 1, pp. 260-267, September 1975.

120. Shortliffe, Edward H., Davis, Randall, Axline, Stanton G., Buchanan, Bruce G., Green, C. Cordell, and Cohen, Stanley N., **Computer-Based Consultations in Clinical Therapeutics: Explanation and Rule Acquisition Capabilities of the MYCIN System**, *Computers and Biomedical Research*, Volume 8, Number 3, June 1975, pages 303-320.

121. Smith, David Canfield, Horace J. Enea, **Backtracking in MLISP2**, *Proceedings of the Third International Joint Conference on Artificial Intelligence*, Stanford University, August 1973.

122. Smith, Leland, **Editing and Printing Music by Computer**, *J. Music Theory*, Fall 1973.

123. Sobel, Irwin, **On Calibrating Computer Controlled Cameras for Perceiving 3-D Scenes**, *Proc. Third Int. Joint Conf. on Artificial Intelligence*, Stanford U., 1973; also in Artificial Intelligence J., Vol. 5, No. 2, Summer 1974.

124. Suzuki, N., **Verifying Programs by Algebraic and Logical Reduction**, *Proc. Int. Conf. on Reliable Software*, Los Angeles, Calif., April 1975, in *ACM SIGPLAN Notices*, Vol. 10, No. 6, pp. 473-481, June 1975.

125. Tesler, Lawrence G., Horace J. Enea, David C. Smith, **The LISP70 Pattern Matching System**, *Proceedings of the Third International Joint Conference on Artificial Intelligence*, Stanford University, August 1973.

126. Thomas, Arthur J., **Puccetti on Machine Pattern Recognition**, *Brit. J. Philosophy of Science*, 26:227-232, 1975.

127. Veverka, J., Carl Sagan, Lynn Quam, R. Tucker, B. Eross, **Variable Features on Mars III: Comparison of Mariner 1969 and Mariner 1971 Photography**, *Icarus*, 21, 317-368, 1974.

128. von Henke, F. W., D.C. Luckham, **A Methodology for Verifying Programs**, *Proc. Int. Conf. on Reliable Software*, Los Angeles, Calif., April 1975, in *ACM SIGPLAN Notices*, Vol. 10, No. 6, pp. 156-164, June 1975.

129. Wilks, Yorick, **The Stanford Machine Translation and Understanding Project**, in R. Rustin (ed.), *Natural Language Processing*, Algorithmics Press, New York, 1973.

130. Wilks, Yorick, **Understanding Without Proofs**, *Proceedings of the Third International Joint Conference on Artificial Intelligence*, Stanford University, August 1973.

131. Wilks, Yorick, Annette Herskovits, **An Intelligent Analyser and Generator of Natural Language**, *Proc. Int. Conf. on Computational Linguistics*, Pisa, Italy, *Proceedings of the Third Internation Joint Conference on Artificial Intelligence*, Stanford University, August 1973.

132. Wilks, Yorick, **The Computer Analysis of Philosophical Arguments**, *CIRPHO*, Vol. 1, No. 1, September 1973

133. Wilks, Yorick, **An Artificial Intelligence Approach to Machine Translation**, in Schank and Colby (eds.), *Computer Models of Thought and Language*, W. H. Freeman, San Francisco, 1973.

134. Wilks, Yorick, **One Small Head – Models and Theories in Linguistics**, *Foundations of Language*, Vol. 10, No. 1,
80. McCarthy, John, Book Review: **Artificial**

135. Wilks, Yorick, **Preference Semantics**, E. Keenan (ed.), *Proc.* 1973 *Colloquium on Formal Semantics of Natural Language*, Cambridge, U.K., 1974.

136. Wilks, Yorick, **The XGP Computer-driven Printer at Stanford**, *Bulletin of Assoc. for Literary and Linguistic Computing*, , Vol. 2, No. 2, Summer 1974.

137. Wilks, Y., **Semantic Procedures and Information**, in *Studies in the Foundations of Communication*, R. Posner (ed.), Springer, Berlin, forthcoming.

138. Wilks, Yorick, **A Preferential, Pattern-Seeking Semantics for Natural Language Inference**, *Artificial Intelligence J.*, Vol. 6, No. 1, Spring 1975.

139. Wilks, Y., **An Intelligent Analyser and Understander of English**, *Comm. ACM*, May 1975.

140. Winograd, Terry, **A Process Model of Language Understanding**, in Schank and Colby (eds.), *Computer Models of Thought and Language*, W. H. Freeman, San Francisco, 1973.

141. Winograd, Terry, **The Processes of Language Understanding** in Benthall, (ed.), *The Limits of Human Nature*, Allen Lane, London, 1973.

142. Winograd, Terry, **Language and the Nature of Intelligence**, in G.J. Dalenoort (ed.), *Process Models for Psychology*, Rotterdam Univ. Press, 1973

143. Winograd, Terry, **Breaking the Complexity Barrier (again)**, *Proc. SIGPLAN–SIGIR Interface Meeting*, 1973; *ACM SIGPLAN Notices*, 10:1, pp. 13–30, January 1975.

144. Winograd, Terry, **Artificial Intelligence – When Will Computers Understand People?**, *Psychology Today*, May 1974.

145. Winograd, Terry, **Frame Representations and the Procedural - Declarative Controversy**, in D. Bobrow and A. Collins, eds., *Representation and Understanding: Studies in Cognitive Science*, Academic Press, 1975.

146. Winograd, Terry, **Reactive Systems**, *Coevolution Quarterly*, September 1975

147. Winograd, Terry, **Parsing Natural Language via Recursive Transition Net**, in Raymond Yeh (ed.) *Applied Computation Theory*, Prentice-Hall, 1976.

148. Winograd, Terry, **Computer Memories – a Metaphor for Human Memory**, in Charles Cofer (ed.), *Models of Human Memory*, Freeman, 1976.

149. Yakimovsky, Yoram, Jerome A. Feldman, **A Semantics-Based Decision Theoretic Region Analyzer**, *Proceedings of the Third International Joint Conference on Artificial Intelligence*, Stanford University, August 1973.

150. Yolks, Warwick, **There's Always Room at the Top, or How Frames gave my Life Meaning**, *SIGART Newsletter*, No. 53, August 1975.

Appendix D
Abstracts of Recent Reports

Abstracts are given here for Artificial Intelligence Memos published since 1976. For earlier years, see our ten-year report [Memo AIM-228] or diskfile AIMS.OLD [BIB,DOC] @SU-AI. The abstracts below are kept in diskfile AIMS [BIB,DOC] @SU-AI and the titles of both earlier and more recent A. I. Memos are in AIMLST[BIB,DOC] @SU-AI.

In the listing below, there are up to three numbers given for each report: an "AIM" number on the left, a "CS" (Computer Science) number in the middle, and a NTIS stock number (often beginning "AD...") on the right. Special symbols preceding the "AIM" number indicate availability at this writing, as follows:

+ hard copy or microfiche,
@ microfiche only,
* out-of-stock.

If there is no special symbol, then it is available in hard copy only. Reports that are in stock may be requested from:

Documentation Services
Artificial Intelligence Laboratory
Stanford University
Stanford, California 94305

Rising costs and restrictions on the use of research funds for printing reports have made it necessary to charge for reports at their replacement cost. By doing so, we will be able to reprint popular reports rather than simply declaring them "out of print".

Alternate Sources

Alternatively, reports may be ordered (for a nominal fee) in either hard copy or microfiche from:

National Technical Information Service
P. O. Box 1553
Springfield, Virginia 22161

If there is no NTIS number given, then they may or may not have the report. In requesting copies in this case, give them both the "AIM-" and "CS-nnn" numbers, with the latter enlarged into the form "STAN-CS-yy-nnn", where "yy" is the last two digits of the year of publication.

Memos that are also Ph.D. theses are so marked below and may be ordered from:

University Microfilm
P. O. Box 1346
Ann Arbor, Michigan 48106

For people with access to the ARPA Network, the texts of some A. I. Memos are stored online in the Stanford A. I. Laboratory disk file. These are designated below by "Diskfile: <file name>" appearing in the header.

@ AIM-277 CS-542 ADA027454
Zohar Manna, Adi Shamir,
The Theoretical Aspects of the Optimal Fixedpoint,
24 pages, March 1976.

In this paper we define a new type of fixedpoint of recursive definitions and investigate some of its properties. This optimal fixedpoint (which always uniquely exists) contains, in some sense, the maximal amount of "interesting" information which can be extracted from the recursive definition, and it may be strictly more defined than the program's least fixedpoint. This fixedpoint can be the basis for assigning a new semantics to recursive programs.

+ AIM-278 CS-549 ADA027455
David Luckham, Norihisa Suzuki,
Automatic Program Verification V: Verification-Oriented Proof Rules for Arrays, Records and Pointers,
48 pages, March 1976. Cost: $3.05

A practical method is presented for automating in a uniform way the verification of Pascal programs that operate on the standard Pascal data structures ARRAY, RECORD, and POINTER. New assertion language primitives are introduced for

describing computational effects of operations on these data structures. Axioms defining the semantics of the new primitives are given. Proof rules for standard Pascal operations on pointer variables are then defined in terms of the extended assertion language. Similar rules for records and arrays are special cases. An extensible axiomatic rule for the Pascal memory allocation operation, NEW, is also given.

These rules have been implemented in the Stanford Pascal program verifier. Examples illustrating the verification of programs which operate on list structures implemented with pointers and records are discussed. These include programs with side-effects.

⊛ AIM-279 CS-552
Norihsa Suzuki,
Automatic Verification of Programs with Complex Data Structures,
Thesis: Ph.D. in Computer Science,
194 pages, February 1976.

The problem of checking whether programs work correctly or not has been troubling programmers since the earliest days of computing. Studies have been conducted to formally define semantics of programming languages and derive proof rules for correctness of programs.

Some experimental systems have been built to mechanically verify programs based on these proof rules. However, these systems are yet far from attacking real programs in a real environment. Many problems covering the ranges from theory to artificial intelligence and programming languages must be solved in order to make program verification a practical tool. First, we must be able to verify a complete practical programming language. One of the important features of real programming languages which is not treated in early experimental systems is complex data structures. Next, we have to study specification methods. In order to verify programs we have to express what we intend to do by the programs. In many cases we are not sure what we want to verify and how we should express them. These specification methods are not independent of the proof rules. Third, we have to construct an efficient prover so that we can interact with the verification process. It is expected that repeated verification attempts will be necessary because programs and specifications may have errors at first try. So the time to complete one verification attempt is very important in real environment.

We have chosen Pascal as the target language. The semantics and proof rules are studied by Hoare & Wirth and Igarashi, London & Luckham. However, they have not treated complex data structures obtained from arrays, records, and pointers. In order to express the state of the data structures concisely and express the effects of statements we introduced special assertion language primitives and new proof rules. We defined new methods of introducing functions and predicates to write assertions so that we can express simplification rules and proof search strategies. We introduced a special language to document properties of these functions and predicates. These methods enable users to express assertions in natural ways so that verification becomes easier. The theorem prover is constructed so that it will be efficient for proving a type of formulas which appear very often as verification conditions.

We have successfully verified many programs. Using our new proof rules and specification methods we have proved properties of sorting programs such as permutation and stability which have been thought to be hard to prove. We see no theoretical as well as practical problems in verifying sorting programs. We have also verified programs which manipulate pointers. These programs change their data structures so that usually verification conditions tend to be complex and hard to read. Some study about the complexity problem seems necessary.

The verifier has been used extensively by various users, and probably the most widely used verifier implemented so far. There is yet a great deal of research necessary in order to fill the gap between the current verifier and the standard programming tools like editors and compilers.

This dissertation was submitted to the Department of Computer Science and the January 1974.
University in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

+ AIM-280 CS-555
David D. Grossman,
Monte Carlo Simulation of Tolerancing in Discrete Parts Manufacturing and Assembly,
25 pages, May 1976. Cost: $2.40

The assembly of discrete parts is strongly affected by imprecise components, imperfect fixtures and tools, and inexact measuremets. It is often necessary to design higher precision into the manufacturing and assembly process than is functionally needed in the final product. Production engineers must trade off between alternative ways of selecting individual tolerances in order to achieve minimum cost, while preserving product integrity. This paper describes a comprehensive Monte Carlo method for systematically analysing the stochastic implications of tolerancing and related forms of imprecision. The method is illustrated by four examples, one of which is chosen from the field of assembly by computer controlled manipulators.

+ AIM-281.1 CS-558 AD-A042 507
Zohar Manna, Richard Waldinger,
Is 'sometime' sometimes better than 'always'? Intermittent assertions in proving program correctness,
41 pages, June 1976, revised March 1977.
Cost: $2.85

This paper explores a technique for proving the correctness and termination of programs simultaneously. This approach, which we call the [intermittent]-[assertion method], involves documenting the program with assertions that must be true at some time when control is passing through the corresponding point, but that need not be true every time. The method, introduced by Knuth and further developed by Burstall, promises to provide a valuable complement to the more conventional methods.

We first introduce and illustrate the technique with a number of examples. We then show that a correctness proof using the invariant assertion method or the subgoal induction method can always be expressed using intermittent assertions instead, but that the reverse is not always the case. The method can also be used just to prove termination, and any proof of termination using the conventional well-founded sets approach can be rephrased as a proof using intermittent assertions. Finally, we show how the method can be applied to prove the validity of program transformations and the correctness of continuously operating programs.

This is a revised and simplified version of a pevious paper with the same title (AIM-281, June 1976).

+ AIM-282 CS-560
Russell Taylor,
Synthesis of Manipulator Control Programs from Task-level Specifications,
Thesis: Ph.D. in Computer Science,
229 pages, July 1976. Cost: $8.10

This research is directed towards automatic generation of manipulator control programs from task-level specifications. The central assumption is that much manipulator-level coding is a process of adapting known program constructs to particular tasks, in which coding decisions are made by well-defined computations based on *planning information*. For manipulator programming, the principal elements of planning information

are: (1) descriptive information about the objects being manipulated; (2) situational information describing the execution-time environment; and (3) action information defining the task and the semantics of the execution-time environment.

A standard subtask in mechanical assembly, insertion of a pin into a hole, is used to focus the technical issues of automating manipulator coding decisions. This task is first analyzed from the point of view of a human programmer writing in the target language, AL, to identify the specific coding decisions required and the planning information required to make them. Then, techniques for representing this information in a computationally useful form are developed. Objects are described by attribute graphs, in which the nodes contain shape information, the links contain structural information, and properties of the links contain location information. Techniques are developed for representing object locations by parameterized mathematical expressions in which free scalar variables correspond to degrees of freedom and for deriving such descriptions from symbolic relations between object features. Constraints linking the remaining degrees of freedom are derived and used to predict maximum variations. Differential approximations are used to predict errors in location values. Finally, procedures are developed which use this planning information to generate AL code automatically.

The AL system itself performs a number of coding functions not normally found in algebraic compilers. These functions and the planning information required to support them are also discussed.

⊛ AIM-283 CS-552
Randall Davis,
Applications of Meta Level Knowledge to the Construction, Maintenance and Use of Large Knowledge Bases,
Thesis: Ph.D. in Computer Science,
304 pages, July 1976.

The creation and management of large knowledge bases has become a central problem of artificial intelligence research as a result of two recent trends: an emphasis on the use of large stores of domain specific knowledge as a base for high performance programs, and a concentration on problems taken from real world settings. Both of these mean an emphasis on the accumulation and management of large collections of knowledge, and in many systems embodying these trends much time has been spent on building and maintaining such knowledge bases. Yet there has been little discussion or analysis of the concomitant problems. This thesis attempts to define some of the issues involved, and explores steps taken toward solving a number of the problems encountered. It describes the organization, implementation, and operation of a program called TEIRESIAS, designed to make possible the interactive transfer of expertise from a human expert to the knowledge base of a high performance program, in a dialog conducted in a restricted subset of natural language.

The two major goals set were (*i*) to make it possible for an expert in the domain of application to "educate" the performance program directly, and (*ii*) to ease the task of assembling and maintaining large amounts of knowledge.

The central theme of this work is the exploration and use of what we have labelled *meta level knowledge*. This takes several different forms as its use is explored, but can be summed up generally as "knowing what you know". It makes possible a system which has both the capacity to use its knowledge directly, and the ability to examine it, abstract it, and direct its application.

We report here on the full extent of the capabilities it makes possible, and document cases where its lack has resulted in significant difficulties. Chapter 3 describes efforts to enable a program to explain its actions, by giving it a model of its control structure and

an understanding of its representations. Chapter 5 documents the use of abstracted models of knowledge (*rule models*) as a guide to acquisition. Chapter 6 demonstrates the utility of describing to a program the structure of its representations (using *data structure schemata*). Chapter 7 describes the use of strategies in the form of *meta rules*, which contain knowledge about the use of knowledge.

⊛ AIM-284 CS-567
Rafael Finkel,
Constructing and Debugging Manipulator Programs,
Thesis: Ph.D. in Computer Science,
171 pages pages, August 1976.

This thesis presents results of work done at the Stanford Artificial Intelligence Laboratory in the field of robotics. The goal of the work is to program mechanical manipulators to accomplish a range of tasks, especially those found in the context of automated assembly. The thesis has three chapters describing significant work in this domain. The first chapter is a textbook that lays a theoretical framework for the principal issues involved in computer control of manipulators, including types of manipulators, specification of destinations, trajectory specification and planning, methods of interpolation, force feedback, force application, adaptive control, collision avoidance, and simultaneous control of several manipulators. The second chapter is an implementation manual for the AL manipulator programming language. The goals of the language are discussed, the language is defined, the compiler described, and the execution environment detailed. The language has special facilities for condition monitoring, data types that represent coordinate systems, and affixment structures that allow coordinate systems to be linked together. Programmable side effects play a large role in the implementation of these features. This chapter closes with a detailed programming example that displays how the constructs of the language assist in formulating and encoding the manipulation task. The third chapter discusses the problems involved in programming in the AL language, including program preparation, compilation, and especially debugging. A debugger, ALAID, is designed to make use of the complex environment of AL. Provision is made to take advantage of the multiple-processor, multiple-process, real-time, interactive nature of the problem. The principal conclusion is that the debugger can fruitfully act as a uniform supervisor for the entire process of program preparation and as the means of communication between cooperating processors.

⊛ AIM-285 CS-568 PB-259 130/3WC
T. O. Binford, D. D. Grossman, C. R. Lui, R. C. Bolles, R. A. Finkel, M. S. Mujtaba, M. D. Roderick, B. E. Shimano, R. H. Taylor, R. H. Goldman, J. P. Jarvis, V. D. Scheinman, T. A. Gafford,
Exploratory Study of Computer Integrated Assembly Systems, Progress Report 3,
336 pages, August 1976.

The Computer Integrated Assembly Systems project is concerned with developing the software technology of programmable assembly devices, including computer controlled manipulators and vision systems. A complete hardware system has been implemented that includes manipulators with tactile sensors and TV cameras, tools, fixtures, and auxiliary devices, a dedicated minicomputer, and a time-shared large Committee on Graduate Studies of Stanford terminals. An advanced software system call AL has been developed that can be used to program assembly applications. Research currently underway includes refinement of AL, development of improved languages and interactive programming techniques for assembly and vision, extension of computer vision to areas which are currently infeasible, geometric modeling of objects and constraints, assembly simulation, control algorithms, and adaptive methods of calibration.

+ AIM-285.4 CS-568 PB-259 130/3WC
T. O. Binford, C. R. Lui, G. Gini, M. Gini, I. Glaser, T. Ishida, M. S. Mujtaba, E. Nakano, H. Nabavi, E. Panofsky, B. E. Shimano, R. Goldman, V. D. Scheinman, D. Schmelling, T. A. Gafford,
Exploratory Study of Computer Integrated Assembly Systems, Progress Report 4,
255 pages, June 1977. Cost: $8.85

The Computer Integrated Assembly Systems project is concerned with developing the software technology of programmable assembly devices. A primary part of the research has been designing and building the AL language for assembly. A first level version of AL is now implemented and debugged, with user interfaces. Some of the steps involved in completing the system are described. The AL parser has been completed and is documented in this report. A preliminary interface with vision is in operation. Several hardware projects to support software development have been completed. One of the two Stanford arms has been rebuilt. An electronic interface for the other arm has been completed. Progress on other hardware aspects of the AL systems is reported.

Several extensions to AL are described. A new interactive program for building models by teaching is running and undergoing further development. Algorithms for force compliance have been derived; a software system for force compliance has been implemented and is running in the AL runtime system. New algorithms have been derived for cooperative manipulation using two manipulators. Preliminary results are described for parth calculation; these results are steps along the way to a runtime path calculator which will be important in making an export version of AL.

Results are described in analysis of several complex assemblies. These results show that two manipulators are necessary in a significant fraction of assembly operations.

Studies are described which focus on making AL meet the realities of industrial research and production use. Results of a questionaire of leading industrial and research laboratores are presented. A summary is presented of the Workshop on Softwre for Assembly, held immediately before the NSF-RANN Conference at IITRI, Nov. 1976.

⊛ AIM-286 CS-570
Douglas Lenat,
AM: An Artificial Intelligence Approach to Discovery in Mathematics as Heuristic Search,
Thesis: Ph.D. in Computer Science,
350 pages, July 1976.

A program, called "AM", is described which models one aspect of elementary mathematics research: developing new concepts under the guidance of a large body of heuristic rules. "Mathematics" is considered as a type of intelligent behavior, not as a finished product.

+ AIM-287 CS-571
Michael Roderick,
Discrete Control of a Robot Arm,
Thesis: Engineer in Electrical Engineering,
98 pages, August 1976. Cost: $4.45

The primary goal of this thesis was to determine the feasibility of operating the Stanford robot arm and reduce sample rates. A secondary goal was to reduce the effects of variations in inertia and sampling rates on the control system's stability.

A discrete arm model was initially developed to illustrate the effects of inertia and sampling rate variations on the present control system. Modifications were then suggested for reducing these effects. Finally, a method was demonstrated for reducing the arm sampling rate from its present value of 60 hertz to approximately 45 hertz without significantly effecting the arms performance.

+ AIM-288 CS-572
Robert Filman, Richard Weyhrauch,
An FOL Primer,
36 pages, September 1976. Cost: $2.70

This primer is an introduction to FOL, an interactive proof checker for first order logic. Its examples can be used to learn the FOL system, or read independently for a flavor of our style of interactive proof checking. Several example proofs are presented, successively increasing in the complexity of the FOL commands employed.

FOL runs on the computer at the Stanford Artificial Intelligence Laboratory. It can be used over the ARPA net after arrangements have been made with Richard Weyhrauch (network address RWW@SU-AI).

+ AIM-289 CS-574
John Reiser (ed.),
SAIL,
178 pages, August 1976. Cost: $6.70

SAIL is a high-level programming language for the PDP-10 computer. It includes an extended ALGOL 60 compiler and a companion set of execution-time routines. In addition to ALGOL, the language features: (1) flexible linking to hand-coded machine language algorithms, (2) complete access to the PDP-10 I/O facilities, (3) a complete system of compile-time arithmetic and logic as well as a flexible macro system, (4) a high-level debugger, (5) records and references, (6) sets and lists, (7) an associative data structure, (8) independent processes, (9) procedure variables, (10) user modifiable error handling, (11) backtracking, and (12) interrupt facilities.

This manual describes the SAIL language and the execution-time routines for the typical SAIL user: a non-novice programmer with some knowledge of ALGOL. It lies somewhere between being a tutorial and a reference manual.

+ AIM-290 CS-575 AD-A042 494
Nancy W. Smith,
SAIL Tutorial,
54 pages, November 1976. Cost: $3.20

This TUTORIAL is designed for a beginning user of Sail, an ALGOL-like language for the PDP10. The first part covers the basic statements and expressions of the language; remaining topics include macros, records, conditional compilation, and input/output. Detailed examples of Sail programming are included throughout, and only a minimum of programming background is assumed.

⊗ AIM-291 CS-577 AO44713
Bruce Buchanan, Joshua Lederberg, John McCarthy,
Three Reviews of J. Weizenbaum's Computer Power and Human Reason,
28 pages, November 1976.

Three reviews of Joseph Weizenbaum's **Computer Power and Human Reason** (W.H. Freeman and Co., San Francisco, 1976) are reprinted from other sources. A reply by Weizenbaum to McCarthy's review is also reprinted.

+ AIM-292 CS-580
Terry Winograd,
Towards a Procedural Understanding of Semantics,
30 pages, October 1976. Cost: $2.55

The term "procedural semantics" has been used in a variety of ways, not all compatible, and not all comprehensible. In this paper, I have chosen to apply the term to a broad paradigm for studying semantics (and in fact, all of linguistics). This paradigm has developed in a context of writing computer programs which use natural language, but it is not a theory of computer programs or programming techniques. It is "procedural" because it looks at the underlying structure of language as fundamentally shaped by the nature of processes for language production and comprehension. It is based on the belief

that there is a level of explanation at which there are significant similarities between the psychological processes of human language use and the computational processes in computer programs we can construct and study. Its goal is to develop a body of theory at this level. This approach necessitates abandoning or modifying several currently accepted doctrines, including the way in which distinctions have been drawn between "semantics" and "pragmatics" and between "performance" and "competence".

The paper has three major sections. It first lays out the paradigm assumptions which guide the enterprise, and elaborates a model of cognitive processing and language use. It then illustrates how some specific semantic problems might be approached from a procedural perspective, and contrasts the procedural approach with formal structural and truth conditional approaches. Finally, it discusses the goals of linguistic theory and the nature of the linguistic explanation.

Much of waht is presented here is a speculation about the nature of a pradigm yet to be developed. This paper is an attempt to be evocative rather than definitive; to convey intuitions rather than to formulate crucial arguments which justify this approach over others. It will be successful if it suggests some ways of looking at language which lead to further understanding.

⊛ AIM-293 CS-581 AD-A042 508
Daniel Bobrow, Terry Winograd,
An Overview of KRL,
40 pages, November 1976.

This paper describes KRL, a Knowledge Representation Language designed for use in understander systems. It outlines both the general concepts which underlie our research and the details of KRL-0, an experimental implementation of some of these concepts. KRL is an attempt to integrate procedural knowledge with a broad base of declarative forms. These forms provide a variety of ways to express the logical structure of the knowledge, in order to give flexibility in associating procedures (for memory and reasoning) with specific pieces of knowledge, and to control the relative accessibility of different facts and descriptions. The formalism for declarative knowledge is based on *structured conceptual objects* with associated *descriptions*. These objects form a network of *memory units* with several different sorts of linkages, each having well-specified implications for the retrieval process. Procedures can be associated directly with the internal structure of a conceptual object. This *procedural attachment* allows the steps for a particular operation to be determined by characteristics of the specific entities involved.

The control structure of KRL is based on the belief that the next generation of intelligent programs will integrate data-directed and goal-directed processing by using multi-processing. It provides for a priority-ordered multi-process agenda with explicit (user-computer equipped with graphic display resource allocation. It provides *procedure directories* which operate along with *process frameworks* to allow procedural parameterization of the fundamental system processes for building, comparing, and retrieving memory structures. Future development of KRL will include integrating procedure definition with the descriptive formalism.

+ AIM-294 CS-586 AD-A042 516
Nachum Dershowitz, Zohar Manna,
The Evolution of Programs: A System for Automatic Program Modification,
45 pages, December 1976. Cost: $2.95

An attempt is made to formulate techniques of program modification, whereby a program that achieves one result can be transformed into a new program that uses the same principles to achieve a different goal. For example, a program that uses the binary search paradigm to calculate the square-root of a number may be modified to divide two numbers in a similar manner, or vice versa.

Program debugging is considered as a special case of modification: if a program computes wrong results, it must be modified to achieve the intended results. the application of abstract program schemata to concrete problems is also viewed from the perspective of modification techniques.

We have embedded this approach in a running implementation; our methods are illustrated with several examples that have been performed by it.

+ AIM-295 CS-591
Robert C. Bolles,
Verification Vision Within a Programmable Assembly System,
Thesis: Ph.D. in Computer Science,
245 pages, December 1976. Cost: $8.55

The long-range goal of this research is to simplify visual information processing by computer. The research reported in this thesis concentrates on a subclass of visual information processing referred to as *verification vision* (abbreviated VV). VV includes a significant portion of the visual feedback tasks required within programmable assembly. There are several types of information available in VV tasks that can facilitate the solution of such tasks. The main question addressed in this thesis is how to use all of this information to perform the task efficiently. Two steps are involved in answering this question: (1) formalize the types of tasks, available information, and quantities of interest and (2) formulate combination rules that use the available information to estimate the quantities of interest.

The combination rules that estimate confidences are based upon Bayes' theorem. They are general enough to handle operators that are not completely reliable, i.e., operators that may find any one of several features or a *surprise*. The combination rules that estimate precisions are based upon a least-squares technique. They use the expected precisions of the operators to check the structural consistency of a set of matches and to estimate the resulting precisions about the points of interest. An interactive VV system based upon these ideas has been implemented. It makes it possible for a person who is not an expert in vision research to program visual feedback tasks. This system helps the programmer select potentially useful operator/feature pairs, provides a training session to gather statistics on the behavior of the operators, automatically ranks the operator/feature pairs according to their expected contributions, and performs the desired task. The VV system has also been interfaced to the AL control system for the mechanical arms and has been tested on tasks that involve a combination of touch, force, and visual feedback.

+ AIM-296 CS-592
Robert Cartwright,
Practical Formal Semantic Definition and Verification Systems,
Thesis: Ph.D. in Computer Science,
158 pages, December 1976. Cost: $6.15

Despite the fact that computer scientists have developed a variety of formal methods for proving computer programs correct, the formal verification of a non-trivial program is still a formidable task. Moreover, the notion of proof is so imprecise in most existing verification systems, that the validity of the proofs generated is open to question. With an aim toward rectifying these problems, the research discussed in this dissertation attempts to accomplish the following objectives:

1. To develop a programming language which is sufficiently powerful to express many interesting algorithms clearly and succintly, yet simple enough to have a tractable formal semantic definition.

2. To completely specify both proof theoretic and model theoretic formal semantics for this language using the simplest possible abstractions.

3. To develop an interactive program verification system for the language which automatically performs as many of the straightforward steps in a verification as possible. →[continued next page] .univ .next page

The first part of the dissertation decribes the motivation for creating TYPED LISP, a variant of PURE LISP including a flexible data type definition facility allowing the programmer to create arbitrary recursive types. It is argued that a powerful data type definition facility not only simplifies the task of writing programs, but reduces the complexity of the complementary task of verifying those programs.

The second part of the thesis formally defines the semantics of TYPED LISP. Every function symbol defined in a program P is identified with a function symbol in a first order predicate calculus language Lp. Both a standard model Mp and a natural deduction system Np are defined for the language Lp. In the standard model, each function symbol is interpreted by the least call-by-value fixed-point of its defining equation. An informal meta-mathematical proof of the consistency of the model Mp and the deductive system Np is given.

The final part of the dissertation describes an interactive verification system implementing the natural deduction system Np.

The verification system includes:

1. A subgoaler which applies rules specified by the user to reduce the proof of the current goal (or theorem) to the proof of one or more subgoals.

2. A powerful simplifier which automatically proves many non-trivial goals by utilizing user-supplied lemmas as well as the rules of Np.

With a modest amount of user guidance, the verification system has proved a number of interesting, non-trivial theorems including the total correctness of an algorithm which sorts by successive merging, the total correctness of the McCarthy-Painter compiler for expressions, the termination of a unification algorithm and the equivalence of an iterative algorithm and a recursive algorithm for counting the leafs of a tree. Several of these proofs are included in an appendix.

⊛ AIM-297 CS-610
Terry Winograd,
A Framework for Understanding Discourse,
24 pages, April 1977.

There is a great deal of excitement in linguistics, cognitive psychology, and artificial intelligence today about the potential of understanding discourse. Researchers are studying a group of problems in natural language which have been largely ignored or finessed in the mainstream of language research over the past fifteen years. They are looking into a wide variety of phenomena, and although results and observations are scattered, it is apparent that there are many interrelationships. While the field s not yet at a stage where it is possible to lay out a precise unifying theory, this paper attempts to provide a beginning framework for studying discourse. Its main goal is to establish a general context and give a feeling for the problems through examples and references. Its four sections attempt to:

Delimit the range of problems covered by the term "discourse."

Characterize the basic structure of natural language based on a notion of communication.

Propose a general approach to formalisms for describing the phenomena and building theories about them

Lay out an outline of the different schemas involved in generating and comprehending language

⊛ AIM-298 CS-611 ADA046703
Zohar Manna, Richard Waldinger,
The Logic of Computer Programming,
90 pages, June 1977.

Techniques derived from mathematical logic promise to provide an alternative to the conventional methodology for constructing, debugging, and optimizing computer programs. Ultimately, these techniques are intended to lead to the automation of many of the facets of the programming process.

In this paper, we provide a unified tutorial exposition of the logical techniques, illustrating each with examples. We assess the strengths and limitations of each technique as a practical programming aid and report on attempts to implement these methods in experimental systems.

+ AIM-299 CS-614 ADA049760
Zohar Manna, Adi Shamir,
The Convergence of Functions to Fixedpoints of Recursive Definitions,
45 pages, May 1977. Cost: $2.95

The classical method for constructing the least fixedpoint of a recursive definition is to generate a sequence of functions whose initial element is the totally undefined function and which converges to the desired least fixedpoint. This method, due to Kleene, cannot be generalized to allow the construction of other fixedpoints.

In this paper we present an alternate definition of convergence and a new [fixedpoint access] method of generating sequences of functions for a given recursive definition. The initial function of the sequence can be an arbitrary function, and the sequence will always converge to a fixedpoint that is "close" to the initial function. This defines a monotonic mapping from the set of partial functions onto the set of all fixedpoints of the given recursive definition.

⊛ AIM-300 CS-617
Terry Winograd,
On some Contested Suppositions of Generative Linguistics about the Scientific Study of Language,
25 pages, May 1977.

This paper is a response to a recently published paper which asserts that current work in artificial intelligence is not relevant to the development of theories of language. The authors of that paper declare that workers in AI have misconstrued what the goals of an provided) strategies for scheduling and that there is no reason to believe that the development of programs which could understand language in some domain could contribute to the development of such theories. This paper concentrates on the assumptions underlying their view of science and language. It draws on the notion of "scientific paradigms" as elaborated by Thomas Kuhn, pointing out the ways in which views of what a science should be are shaped by unprovable assumptions. It contrasts the procedural paradigm (within which artificial intelligence research is based) to the currently dominant paradigm typified by the work of Chomsky. It describes the ways in which research in artificial intelligence will increase our understanding of human language, and through an analogy with biology, raises some questions about the plausibility of the Chomskian view of language and the science of linguistics.

+ AIM-301 CS-624 ADA044231
Lester Earnest, et. al.,
Recent Research in Computer Science,
118 pages, June 1977. Cost: $5.00

This report summarizes recent accomplishments in six related areas: (1) basic AI research and formal reasoning, (2) image understanding, (3) mathematical theory of computation, (4) program verification, (5) natural language understanding, and (6) knowledge based programming.

+ AIM-302 CS-630 ADA049761
Zohar Manna, Richard Waldinger
Synthesis: Dreams => Programs,
119 pages, October 1977. Cost: $5.05

Deductive techniques are presented for deriving programs systematically from given specifications. The specifications express the purpose of the desired program without giving any hint of the algorithm to be employed. The basic approach is to transform the specifications repeatedly according to certain rules, until a satisfactory program is produced. The rules are guided by a number of strategic controls. These techniques have been incorporated in a running program synthesis system, called DEDALUS.

Many of the transformation rules represent knowlede about the program's subject domain (e.g. numbers, lists, sets); some represent the meaning of the constructs of the specification language and the target programming language; and a few rules represent basic programming principles. Two of these principles, the conditional-formation rule and the recursion-formation rule, account for the introduction of conditional expressions and of recursive calls into the synthesized program. The termination of the program is ensured as new recursive calls are formed.

Two extensions of the recursion-formation rule are discussed: a procedure-formation rule, which admits the introduction of auxilliary subroutines in the course of the synthesis process, and a generalization rule, which causes the specifications to be extended to represent a more general problem that is nevertheless easier to solve.

The techniques of this paper are illustrated with a sequence of examples of increasing complexity; programs are constructed for list processing, numerical computation, and sorting. These techniques are compared with the methods of "structured programming", and with recent work on "program transformation". The DEDALUS system accepts specifications expressed in a high-level language, including set notation, logical quantification, and a rich vocabulary drawn from a variety of subject domains. The system attempts to transform the specifications into a recursive, LISP-like target program. Over one hundred rules have been implemented, each expressed as a small program in the QLISP language.

⊗ AIM-303 CS-631 ADA050806
Nachum Dershowitz, Zohar Manna,
Inference Rules for Program Annotation,
46 pages, October 1977.

Methods are presented whereby an Algol-like program, given together with its specifications, may be documented automatically. This documentation expresses invariant relationships that hold between program variables at intermediate points in the program, and explains the actual workings of the program regardless of whether the program is correct. Thus this documentation can be used for proving the correctness of the program, or may serve as an aid in the debugging of an incorrect program.

The annotation techniques are formulated as Hoare-like inference rules which derive invariants from the assignment statements, from the control structure of the program, or, heuristically, from suggested invariants. The application of these rules is demonstrated by two examples which have run on our implemented system.

+ AIM-304 CS-632 ADA048684
Todd Wagner,
Hardware Verification,
Thesis: PhD in Computer Science,
102 pages, September 1977. Cost: $4.55

Methods for detecting logical errors in computer hardware designs using symbolic manipulation instead of digital simulation are discussed. A non-procedural register transfer language is proposed that is suitable for describing how a digital circuit should

perform. This language can also be used to describe each of the components used in the design. Transformations are presented which should enable the designer to either prove or disprove that the set of interconnected components correctly satisfy the specifications for the overall system.

The problem of detecting timing anomalies such as races, hazards, and oscillations is addressed. Also explored are some interesting relationships between the problems of hardware verification and program verification. Finally, the results of using an existing proof checking program on some digital circuits are presented. Although the theorem proving approach is not very efficient for simple circuits, it becomes increasingly attractive as circuits become more complex. This is because the theorem proving approach can use complicated component specifications without reducing them to the gate level.

+ AIM-305 CS-633 ADA048660
William Faught,
Motivation and Intensionality in a Computer Simulation Model,
Thesis: Ph.D. in Computer Science,
104 pages, September 1977. Cost: $4.60

This dissertation describes a computer simulation model of paranoia. The model mimics the behavior of a patient participating in a psychiatric interview by answering questions, introducing its own topics, and responding to negatively-valued (e.g., threatening or shame-producing) situations.

The focus of this work is on the motivational mechanisms required to instigate and direct the modelled behavior.

The major components of the model are:

(1) A production system (PS) formalism accounting for the instigation and guidance of behavior as a function of internal (affective) and external (real-world) environmental factors. Each rule in the PS is either an action pattern (AP) or an interpretation pattern (IP). Both may have either affect (emotion) conditions, external variables, or outputs of other patterns as their initial conditions (left-hand sides). The PS activates all rules whose left-hand sides are true, selects the one with the highest affect, and performs the action specified by the right-hand side.

(2) A model of affects (emotions) as an anticipation mechanism based on a small number of basic pain-pleasure factors. Primary activation (raising an affect's strength) occurs when the particular condition for the affect is anticipated (e.g., anticipation of pain for the fear affect). Secondary activation occurs when an internal construct (AP, IP, belief) is used and its associated affect is processed.

(3) A formalism for intensional behavior (directed by internal models) requiring a dual representation of symbol and concept. An intensional object (belief) can be accessed either by sensing it in the environment (concept) or by its name (token). Similarly, an intensional action (intention) can be specified either by its conditions in the immediate environment (concept) or by its name (token).

Issues of intelligence, psychopathological modelling, and artificial intelligence programming are discussed. The paranoid phenomenon is found to be explainable as an extremely skewed use of normal processes. Applications of these constructs are found to be useful in AI programs dealing with error recovery, incompletely specified input data, and natural language specification of tasks to perform.

+ AIM-306 CS-639 ADA053175
Cordell Green, David Barstow,
On Program Synthesis Knowledge,
63 pages, November 1977. Cost: $3.45

This paper presents a body of program

synthesis knowledge dealing with array operations, space reutilization, the divide and conquer paradigm, conversion from recursive paradigms to iterative paradigms, and ordered set enumerations. Such knowledge can be used for the synthesis of efficient and in-place sorts including quicksort, mergesort, sinking sort, and bubble sort, as well as other ordered set operations such as set union, element removal, and element addition. The knowledge is explicated to a level of detail such that it is possible to codify this knowledge as a set of program synthesis rules for use by a computer-based synthesis system. The use and content of this set of programming rules is illustrated herein by the methodical synthesis of bubble sort, sinking sort, quicksort, and mergesort.

+ AIM-307 CS-640 ADA053176
Zohar Manna and Richard Waldinger,
Structured Programming Without Recursion,
10 pages, December 1977. Cost: $2.00

There is a tendency in presentations of structured programming to avoid the use of recursion as a repetitive construct, and to favor instead the iterative loop constructs. For instance, in his recent book, "A Discipline of Programming," Dijkstra bars recursion from his exemplary programming language, declaring that "I don't like to crack an egg with a sledgehammer, no matter how effective the sledgehammer is for doing so."

In introducing an iterative loop, the advocates of structured programming advise that we first find an *invariant assertion* and a *termination function*, and then construct the body of the loop so as to reduce the value of the termination function while maintaining the truth of the invariant assertion. The decision when to introduce a loop, and the choice of an appropriate invariant assertion and termination function, are not dictated by the method, but are left to the intuition of the programmer.

explanatory theory of language should be, and of a program, there are innumerable conditions and functions that could be adopted as the invariant assertion and termination function of a loop. With so many plausible candidates around, a correct selection requires an act of precognitive insight.

As an alternative, we advocate a method of loop formation in which the loop is represented as a recursive procedure rather than as an iterative construct. A recursive procedure is formed when a subgoal in the program's derivation is found to be an instance of a higher-level goal. The decision to introduce the new procedure, its purpose, and the choice of the termination function are all dictated by the structure of the derivation.

The directness of this *recursion-formation* approach stems from the use of recursion rather than iteration as a repetitive construct. Recursion is an ideal vehicle for systematic program construction; in avoiding its use, the advocates of structured programming have been driven to less natural means.

+ AIM-308 CS-641 ADA053184
David Barstow,
Automatic Construction of Algorithms,
Thesis: Ph.D. in Computer Science,
220 pages, December 1977. Cost: $7.85

Despite the wealth of programming knowledge available in the form of textbooks and articles, comparatively little effort has been applied to the codification of this knowledge into machine-usable form. The research reported here has involved the explication of certain kinds of programming knowledge to a sufficient level of detail that it can be used effectively by a machine in the task of constructing concrete implementations of abstract algorithms in the domain of symbolic programming.

Knowledge about several aspects of symbolic programming has been expressed as a collection of four hundred refinement rules.

The rules deal primarily with collections and mappings and ways of manipulating such structures, including several enumeration, sorting and searching techniques. The principle representation techniques covered include the representation of sets as linked lists and arrays (both ordered or unordered), and the representation of mappings as tables, sets of pairs, property list markings, and inverted mappings (indexed by range element). In addition to these general constructs, many low-level programming details are covered (such as the use of variables to store values).

To test the correctness and utility of these rules, a computer system (called PECOS) has been designed and implemented. Algorithms are specified to PECOS in a high-level language for symbolic programming. By repeatedly applying rules from its knowledge base, PECOS gradually refines the abstract specification into a concrete implementation in the target language. When several rules are applicable in the same situation, a refinement sequence can be split. Thus, PECOS can actually construct a variety of different implementations for the same abstract algorithm.

PECOS has successfully implemented algorithms in several application domains, including sorting and concept formation, as well as algorithms for solving the reachability problem in graph theory and for generating prime numbers. PECOS's ability to construct programs from such varied domains suggests both the generality of the rules in its knowledge base and the viability of the knowledge-based approach to automatic programming.

+ AIM-309 CS-646
C. G. Nelson, Derek Oppen,
Efficient Decision Procedures Based on Congruence Closure,
15 pages, January 1978. Cost: $2.15

We define the notion of the *congruence closure* of a relation on a graph and give a simple algorithm for computing it. We then give decision procedures for the quantifier-free theory of equality and the quantifier-free theory of LISP list structure, both based on this algorithm. The procedures are fast enough to be practical in mechanical theorem proving: each procedure determines the satisfiability of a conjunction of length n of literals in time O(n12). We also show that if the axiomatization of the theory of list structure is changed slightly, the problem of determining the satisfiability of a conjunction of literals becomes NP-complete. We have implemented the decision procedures in our simplifier for the Stanford Pascal Verifier.

An earlier version of this paper appeared in the Proceedings of the 18th Annual Symposium on Foundations of Computer Science, Providence, October 1977.

+ AIM-310 CS-651
Nachum Dershowitz, Zohar Manna,
Proving Termination with Multiset Orderings,
33 pages, March 1978. Cost: $2.65

A common tool for proving the termination of programs is the well-founded set, a set ordered in such a way as to admit no infinite descending sequences. The basic approach is to find a termination function that maps the elements of the program into some well-founded set, such that the value of the termination function is continually reduced throughout the computation. All too often, the termination functions required are difficult to find and are of a complexity out of proportion to the program under consideration. However, by providing more sophisticated well-founded sets, the corresponding termination functions can be simplified.

Given a well-founded set S, we consider multisets over S, "sets" that admit multiple occurrences of elements taken from S. We define an ordering on all finite multisets over

S that is induced by the given ordering on S. This multiset ordering is shown to be well-founded.

The value of the multiset ordering is that it permits the use of relatively simple and intuitive termination functions in otherwise difficult termination proofs. In particular, we apply the multiset ordering to provide simple proofs of the termination of production systems, programs defined in terms of sets of rewriting rules.

+ AIM-311 CS652
Greg Nelson, Derek C. Oppen,
Simplification by Cooperating Decision Prodcedures,
20 pages, April 1978. Cost: $2.25

We describe a simplifier for use in program manipulation and verification. The simplifier finds a normal form for any expression over the language consisting of individual variables, the usual boolean connectives, equality, the conditional function cond (denoting if-then-else), the numberals, the arithmetic functions and predicates +, - and ≤, the LISP constants, functions and predicates nil, car, cdr, cons and atom, the functions store and select for storing into and selecting from arrays, and uninterpreted function symbols. Individual variables range over the union of the reals, the set of arrays, LISP list structure and the booleans true and false.

The simplifier is complete; that is, it simplifies every valid formula to true. Thus it is also a decion procedure for the quantifier-free theory of reals, arrays and list structure under the above functions and predicates.

The organization of the simplifier is based on a method for combining decision procedures for several theories into a single decision procedure for a theory combining the original theories. More precisely, given a set S of functions and predicates over a fixed domain, a satisfiability program for S is a program which determines the satisfiability of conjunctions of literals (signed atomic formulas) whose predicate and function symbols are in S. We give a general procedure for combining satisfiability programs for sets S and T into a single satisfiability program for S ∪ T, given certain conditions on S and T.

The simplifier described in this paper is currently used in the Stanford Pascal Verifier.