AD 744 620

DESCRIPTION AND THEORETICAL ANALYSIS (USING SCHEMATA) OF PLANNER: A LANGUAGE FOR PROVING THEOREMS AND MANIPULATING MODELS IN A ROBOT

Carl Hewitt

April 1972

ARTIFICIAL INTELLIGENCE LABORATORY

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Cambridge Massachusetts 02139

UNCLASSIFIED
Security Classification

DOCUMENT CONTROL DATA - R&D

(Security classification of title, body of abstract and indexing annotation must be e ... ered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)* Massachusetts Institute of Technology Artificial Intelligence Laboratory	2a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED 2b. GROUP None
3. REPORT TITLE Description and Theoretical Analysis (using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot	
4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)* Ph.D. Thesis, Department of Mathematics	
5. AUTHOR(S) *(Last name, first name, initial)* Hewitt, Carl	
6. REPORT DATE January 1971	7b. NO. OF REFS 54
8a. CONTRACT OR GRANT NO. N00014-70-A-0362-0003 b. PROJECT NO c. d.	9a. ORIGINATOR'S REPORT NUMBER(S) AI TR-258 9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*
10. AVAILABILITY/LIMITATION NOTICES Distribution of this document is unlimited.	PRICES SUBJECT TO CHANGE
11. SUPPLEMENTARY NOTES None	12. SPONSORING MILITARY ACTIVITY Advanced Research Projects Agency 3D-200 Pentagon Washington, D.C. 20301

13. ABSTRACT

PLANNER is a formalism for proving theorems and manipulating models in a robot. The formalism is built out of a number of problem-solving primitives together with a hierarchical multiprocess backtrack control structure. Statements can be asserted and perhaps later withdrawn as the state of the world changes. Under BACKTRACK control structure, the hierarchy of activations of functions previously executed is maintained so that it is possible to revert to any previous state. Thus programs can easily manipulate elaborate hypothetical tentative states. In addition PLANNER uses multiprocessing so that there can be multiple loci of control over the problem-solving. Conclusions can be drawn from the various changes in state. Goals can be established and dismissed when they are satisfied. The deductive system of PLANNER is subordinate to the hierarchical control structure in order to maintain the desired degree of control. The use of a general-purpose matching language as the basis of the deductive system increases the flexibility of the system. Instead of explicitly naming procedures in calls, procedures can be invoked implicitly by patterns of what the procedure is supposed to accomplish.

14. KEY WORDS

PLANNER	Programming languages	Artificial Intelligence
Problem-solving	Question answering	Semantics
Pattern matching	Theorem proving	Robots
	Schematology	

DD FORM 1473 (M.I.T.)

UNCLASSIFIED
Security Classification

Work reported herein was conducted at the Artificial Intelligence Laboratory, a Massachusetts Institute of Technology research program supported in part by the Advanced Research Projects Agency of the Department of Defense and monitored by the Office of Naval Research under Contract Number N00014-70-A-0362-0003.

DESCRIPTION AND THEORETICAL ANALYSIS (USING SCHEMATA) OF PLANNER: A LANGUAGE FOR PROVING THEOREMS AND MANIPULATING MODELS IN A ROBOT*

Abstract

PLANNER is a formalism for proving theorems and manipulating models in a robot. The formalism is built out of a number of problem-solving primitives together with a hierarchical multiprocess backtrack control structure. Statements can be asserted and perhaps later withdrawn as the state of the world changes. Under BACKTRACK control structure, the hierarchy of activations of functions previously executed is maintained so that it is possible to revert to any previous state. Thus programs can easily manipulate elaborate hypothetical tentative states. In addition PLANNER uses multiprocessing so that there can be multiple loci of control over the problem-solving. Conclusions can be drawn from the various changes in state. Goals can be established and dismissed when they are satisfied. The deductive system of PLANNER is subordinate to the hierarchical control structure in order to maintain the desired degree of control. The use of a general-purpose matching language as the basis of the deductive system increases the flexibility of the system. Instead of explicitly naming procedures in calls, procedures can be invoked implicitly by patterns of what the procedure is supposed to accomplish. The language is being applied to solve problems faced by a robot, to write special purpose routines from goal oriented language, to express and prove properties of procedures, to abstract procedures from protocols of their actions, and as a semantic base for English.

Thesis Supervisor: Seymour Papert, Professor of Mathematics

*This report reproduces a thesis of the same title submitted to the Department of Mathematics, Massachusetts Institute of Technology, on January 29, 1971 in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

0. CONTENTS

0. Contents

1. A Fable on a Declarative Use of Imperatives

2. On the Structural Foundations of Problem Solving

3. Discursive Overview of PLANNER

4. The Pattern Matching Language MATCHLESS

4.1 Syntax of Identifiers and Expressions
4.1.1 Prefix Operators for Identifiers
4.1.2 Expressions

4.2 Types

4.3 Simple examples of Matching

4.4 Definition of Procedures
4.4.1 Functional
4.4.2 Macro
4.4.3 Actor
4.4.4 Type
4.4.4.1 Union
4.4.4.2 Product
4.4.4.3 Extension
4.4.4.4 Sum
4.4.4.5 Homogenous
4.4.5 External Interrupts

4.5 Functions in Expressions
4.5.1 Definitions of Functions
4.5.1.1 Control Functions
4.5.1.1.1 Conditional
4.5.1.1.2 Block
4.5.1.1.3 Escape
4.5.1.1.4 Repetition
4.5.1.1.5 Multi-Process
4.5.1.2 Data Functions

4.5.1.2.1 Specialists
4.5.1.2.1.1 Structural
4.5.1.2.1.1.1 List
4.5.1.2.1.1.2 Vector
4.5.1.2.1.1.3 String
4.5.1.2.1.1.4 Graph
4.5.1.2.1.1.5 Class
4.5.1.2.1.2 Atom
4.5.1.2.1.3 Word and Number
4.5.1.2.1.4 Algebraic
4.5.1.2.1.5 Locative
4.5.1.2.1.6 Stack
4.5.1.2.1.7 Ring
4.5.1.2.1.8 Input-output
4.5.1.2.2 Protection
4.5.1.2.3 Monitoring
4.5.1.2.4 Type
4.5.1.2.5 Synchronization
4.5.1.3 Identifier
4.5.1.4 Debugging
4.5.2 Examples of the Use of Functions

4.6 Actors in Patterns
4.6.1 Definitions of Actors
4.6.1.1 Control Actors
4.6.1.1.1 Conditional
4.6.1.1.2 Block
4.6.1.2 Data Actors
4.6.1.2.1 Specialists
4.6.1.2.1.1 Structural
4.6.1.2.1.1.1 List
4.6.1.2.1.1.2 Vector
4.6.1.2.1.1.3 String
4.6.1.2.1.1.4 Graph
4.6.1.2.1.2 Atom
4.6.1.2.1.3 Word and Number
4.6.1.2.1.4 Algebraic
4.6.1.2.1.5 Locative
4.6.1.2.1.6 Stack
4.6.1.2.1.7 Ring
4.6.1.2.1.8 Character
4.6.1.2.2 Type
4.6.1.3 Identifier
4.6.2 Examples of the Use of Actors

4.7 The Implementation Language BUBBLE

4.8 The Editor

5. The Theorem Proving Formalism PLANNER

5.1 PLANNER Procedures
5.1.1 Hierarchical Backtrack Control Structure
5.1.2 Functional Procedures
5.1.3 Theorems
5.1.3.1 Consequent
5.1.3.2 Antecedent
5.1.3.3 Erasing

5.2 PLANNER Functions
5.2.1 Data Functions
5.2.1.1 Assertions
5.2.1.2 Erasures
5.2.1.3 Goals
5.2.2 Control Functions
5.2.2.1 Failure
5.2.2.2 Finalization
5.2.2.3 Repetition

5.3 Clauses in PLANNER

5.4 A Simple Example
5.4.1 Using a Consequent Theorem
5.4.2 Using an Antecedent theorem
5.4.3 Using Resolution

5.5 Myths about PLANNER
5.5.1 Consequent Theorems Used Only for Working Backwards
5.5.2 PLANNER Does Only Depth First Search
5.5.3 Use of Failure Implies Inefficient Search
5.5.4 PLANNER Does Only What It is Told to Do

6. More on PLANNER

6.1 Simple Examples in PLANNER
6.1.1 Londons's Bridge
6.1.2 Analogies
6.1.2.1 Simple Analogies
6.1.2.2 Structural Analogies
6.1.3 Mathematical Induction
6.1.4 Descriptions
6.1.4.1 Structural Descriptions
6.1.4.2 Constructing Examples of Descriptions
6.1.4.3 Descriptions of Visual Scenes
6.1.4.4 Power Set of Intersection of Two Sets Is the Intersection of Their Power Sets
6.1.5 Semantics of Natural Language

6.1.6 The Pons Asinorum

6.2 Current Problems and Future Work

7. Models of Procedures and the Teaching of Procedures

7.1 Models of Procedures
7.1.1 Models in Expressions: Intentions
7.1.2 Models in Patterns: Aims
7.1.3 Models of PLANNER Theorems

7.2 Teaching procedures
7.2.1 By Telling
7.2.2 By Abstracting Protocols of Procedures
7.2.2.1 Protocols
7.2.2.2 Variatalization and Formation of Protocol Tree
7.2.2.3 Identification of Indistinguishable Nodes
7.2.2.4 Examples
7.2.2.4.1 Building a Wall
7.2.2.4.2 Reversing a List
7.2.2.4.3 Finding the Description of a Stick
7.2.2.4.4 Finding Fibonacci Numbers Iteratively
7.2.2.4.5 Defining a Data Type
7.2.3 By Deducing the Bodies of Canned Loops
7.2.4 Comparison of the Methods

7.3 Current Problems and Future Work

8. Theoretical Foundations

8.1. Analytic Theory
8.1.1 Classes of Schemata
8.1.1.1 Recursive Schemata
8.1.1.1.1 Comparison with Program Schemata
8.1.1.1.2 Compilation
8.1.1.1.3 Schemata with Resets
8.1.1.1.4 Decompilation
8.1.1.1.5 Primitive Recursive Schemata
8.1.1.2 Schemata with Counters
8.1.1.3 Parallel Schemata
8.1.1.4 Locative Schemata
8.1.1.5 Schemata with Selectors and Replacement
8.1.1.6 Schemata with Free Variables

8.1.1.7 Schemata with Equality
8.1.1.8 Hierarchical Backtrack Schemata
8.1.1.8.1 Comparison with Recursive Schemata
8.1.1.8.2 Comparison with Parallel Schemata
8.1.1.8.3 PLANNER Schemata
8.1.2 Intentions
8.1.2.1 Definition of Intentions
8.1.2.2 Completeness of Intentional Analysis

8.2. Synthetic Theory
8.2.1 Realizations
8.2.1.1 Realizations for the Quantificational Calculus
8.2.1.2 Realizations of PLANNER Theorems
8.2.2 Construction of Schemata
8.2.2.1 Completeness of Procedural Abstraction
8.2.2.2 Completeness of Method of Canned Loops

8.3. Current Problems and Future Work

9. Formal Definition of PLANNER

10. Index of Procedures

11. Bibliography

12. Biography

Dedication

This paper is dedicated

to the ideas embodied in the language

LISP

ACKNOWLEDGEMENTS

The following is a report on some of the work that I have done as a graduate student at Project MAC. Reproduction in full or in part is permitted for any purpose of the United States government. Most of the ideas described herein are not original with the author. Many are simple extensions and modifications of current ideas in the computer culture. Others have been suggested by people in conversations. I have tried to explicitly acknowledge all the cases that I can remember My apoligies to any one who has been omitted. Still other ideas have emerged in the course of debate and discussion with the people listed below. I would like to thank the various system "hackers" that have made this work possible: D. Eastlake, R. Greenblatt, J. Holloway, T. Knight, G. Mitchell, S. Nelson, and J. White. I had several useful discussions with H. V. McIntosh and A. Guzman on the subject of pattern matching. S. Papert, T. Winograd, and M. Paterson made suggestions for improving the presentation of the material in this thesis. T. Winograd, P. Winston, and G. Sussman made suggestions for improving PLANNER. Alan Kay, Jeff Rulifson, Nick Pippinger, Eugene Charniak, John McCarthy, Nils Nilson, Richard Fikes, Richard Waldinger, Julian Davies, Bruce Anderson, Jack Dennis, Bob Yates, Danny Bobrow, Warren Teitleman, Richard Stallman, Peter Deutch, and Bob Balzer provided illuminating discussions on some of the fine points. Peter Bishop, Dave Reed, Gary Peskin, Gordon Benedict, Al

Solish, Chris Reeve, Gerald Sussman, Bruce Daniels, Drew McDermott, Jeff Hill, and Dave Cressey have worked on implementations. Ira Goldstein, Peter Bishop, Richard Wong, Steve Zilles, Bruce Daniels, Dave Reed, Gary Peskin, Julian Davies, Gordon Benedict, and Jeff Hill helped me to find bugs in previous versions of this document. I would like to thank the members of my thesis committee (Seymour Papert, Marvin Minsky, and Mike Paterson) for their help and advice. This report represents my current imperfect state of knowldege. The above people are in no way responsible for the kludges, errors, and misunderstandings that remain. Please send comments, criticism, and errata to:

Carl Hewitt
M. I. T. Artificial Intelligence Laboratory
545 Technology Square
Cambridge, Mass.
U. S. A.

Note to the Reader

This paper is organized in what purports to be a logical systematic fashion. The organization makes it difficult to get a quick overview. The reader should not try to read the paper in a linear fashion from cover to cover. If he gets stuck he should "pop up" one level and continue.

"YOU HAVE BEEN WARNED"

There is an index of primitives at the end. There is an index to the syntax after the function READ. The following guide is provided for those readers who are not interested in reading the whole paper. Chapter 1 is a "hack". Chapter 2 gives the epistemological foundations for our approach to problem solving. Chapter 3 is a discursive overview of the rest of the thesis using examples of some features of the problem solving language PLANNER. Many of the important ideas in the thesis are touched on somewhere in the chapter. In chapter 4 we find a detailed explanation of the structural pattern matching language MATCHLESS. Readers who are only peripherally interested in pattern matching need read only sections 4.1, 4.2, 4.3, and 4.4. Chapter 5 begins the systematic explanation of PLANNER. It introduces the primitives, data structure, and control structure of the language. In contrast to the quantificational calculus, the semantics of PLANNER are expressed in terms of the properties of the procedures which define the formalism. In chapter 7 we explain how properties of PLANNER procedures can be expressed and proved in the formalism itself. Also we attack the problem of how it is possible to teach a problem solver new knowledge. We explain how schemata give the beginning of a theory on the comparative problem solving power of various computational models in chapter 8.

1. What Achilles Said To The Tortoise

Lewis Carroll

Achilles had overtaken the Tortoise, and had seated himself comfortably on its back.

"So you've got to the end of our race-course?" said the Tortoise. "Even though it does consist of an infinite series of distances? I thought some wiseacre or other had proved that the thing couldn't be done?"

"It can be done," said Achilles. "It has been done! Solvitur ambulando. You see the distances were constantly diminishing: and so--"

"But if they had been constantly increasing?" the Tortoise interrupted. "How then?"

"Then I shouldn't be here," Achilles modestly replied: "and you would have got several times round the world, by this time!"

"You flatter me-- flatten, I mean," said the Tortoise; "For you are a heavy weight, and no mistake! Well now, would you like to hear of a race-course, that most people fancy they can get to the end of in two or three steps, while it really consists of an infinite number of distances, each one longer than the previous one?"

"Very much indeed!" said the Grecian warrior, as he drew from his helmet (few Grecian warriors possessed pockets in those days) an enormous note-book and a pencil. "Proceed! And speak slowly, please!

Short-hand isn't invented yet!"

"That beautiful First Proposition of Euclid!" the Tortoise murmured dreamily. "You admire Euclid?"

"Passionately! So far, at least, as one can admire a treatise that won't be published for some centuries to come!"

"Well, now, let's take a little bit of the argument in that First Proposition--just two steps, and the conclusion drawn from them. Kindly enter them in your note-book. And, in order to refer to them conveniently, let's call them A, B, and Z:

(A) Things that are equal to the same are equal to each other.

(B) The two sides of this Triangle are things that are equal to the same.

(Z) The two sides of this Triangle are equal to each other.

"Readers of Euclid will grant, I suppose, that Z follows logically from A and B, so that any one who accepts A and B as true, must accept Z as true?"

"Undoubtedly! The youngest child in a High School-- as soon as High Schools are invented, which will not be till some two thousand years later--will grant that."

"And if some reader had not yet accepted A and B as true, he might still accept the Sequence as a valid one, I suppose?"

"No doubt such a reader might exist. He might say 'I accept as true the Hypothetical Proposition that, if A and B be true, Z must be true; but I don't accept A and B as true.' Such a reader would do wisely in abandoning Euclid, and taking to football."

"And might there not also be some reader who would say 'I accept A and B as true, but I don't accept the Hypothetical'?"

"Certainly there might. He, also, had better take to football."

"And neither of these readers," the Tortoise continued, "is as yet under any logical necessity to accept Z as true?"

"Quite so," Achilles assented.

"Well, now, I want you to consider me as a reader of the second kind, and to force me, logically, to accept Z as true."

"A tortoise playing football would be--" Achilles was beginning.

"--an anomaly, of course," the Tortoise hastily interrupted. "don't wander from the point. Let's have Z first, and football afterwards!"

"I'm to force you to accept Z, am I?" Achilles said musingly. "And your present position is that you accept A and B, but you don't accept the Hypothetical--"

"Let's call it C," said the Tortoise.

"--but you don't accept:

(C) If A and B are true, Z must be true."

"That is my present positon," said the Tortoise.

"Then I must ask you to accept C."

"I'll do so," said the Tortoise, "as soon as you've entered it in that note-book of yours. What else have you got in it?"

"Only a few memoranda," said Achilles, nervously fluttering the leaves: "a few memoranda of--of the battles in which I have distinguished myself!"

"Plenty of blank leaves, I see!" the Tortoise cheerily remarked. "We shall need them all!" (Achilles shuddered.) "Now write as I dictate:

(A) Things that are equal to the same are equal each other.

(B) The two sides of this triangle are things that are equal to the same.

(C) If A and B are true, Z must be true.

(Z) The two sides of this Triangle are equal to each other."

"You should call it D, not Z," said Achilles. "It comes next to the other three. If you accept A and B and C, you must accept Z."

"And why must I?"

"Because it follows logically from them. If A and B and C are true, Z must be true. You don't dispute that, I imagine?"

"If A and B and C are true, Z must be true," the Tortoise thoughtfully repeated. "That's another Hypothetical isn't it? And, if I failed to see its truth, I might accept A and B and C, and still not accept Z, mightn't I?"

"You might," the candid hero admitted; "though such obtuseness would certainly be phenomenal. Still, the event is possible. So I must ask you to grant one more Hypothetical."

"Very good. I'm quite willing to grant Z, as soon as you've written it down. We will call it

(D) If A and B and C are true, Z must be true.

"Have you entered that in your note-book?"

"I have!" Achilles joyfully exclaimed, as he ran the pencil into its sheath. "And at last we've got to the end of this ideal race-course! Now that you accept A and B and C and D, of course you accept Z."

"Do I?" said the Tortoise innocently. "Let's make that quite clear. I accept A and B and C and D. Suppose I still refuse to accept Z?"

"Then Logic would take you by the throat, and force you to do it!" Achilles triumphantly replied. "Logic would tell you can't help yourself. Now that you've accepted A and B and C and D, you must accept Z!' So you've no choice, you see."

"Whatever Logic is good enough to tell me is worth writing down," said the Tortoise. "So enter it in your book, please. We will call it

(E) If A and B and C and D are true, Z must be true.

"Until I've granted that, of course, I needn't grant Z. So it's quite a necessary step, you see?"

"I see," said Achilles; and there was a touch of sadness in his tone.

2. The Structural Foundations of Problem Solving

We would like to develop a foundation for problem solving analogous in some ways to the currently existing foundations for mathematics. Thus we need to analyze the structure of foundations for mathematics. A foundation for mathematics must provide a definitional formalism in which mathematical objects can be defined and their existence proved. For example set theory as a foundation provides that objects must be built out of sets. Then there must be a deductive formalism in which fundamental truths can be stated and the means provided to deduce additional truths from those already established. Current mathematical foundations such as set theory seem quite natural and adequate for the vast body of classical mathematics. The objects and reasoning of most mathematical domains such as analysis and algebra can be easily founded on set theory. The existence of certain astronomically large cardinals poses some problems for set theoretic foundations. However, the problems posed seem to be of practical importance only to certain category theorists. Foundations of mathematics have devoted a great deal of attention to the problems of consistency and completeness. The problem of consistency is important since if the foundations are inconsistent then any formula whatsoever may be deduced, thus trivializing the foundations. Semantics for foundations of mathematics are defined model theoretically in terms of the notion of satisfiability. The problem of completeness, is that for a foundation of mathematics to be intuitively satisfactory all the true formulas should be proveable since a foundation for mathematics aims to be a theory of mathematical truth.

Similar fundamental questions must be faced by a foundation for problem solving. However there are some important differences since a foundation for problem solving aims more to be a theory of actions and purposes than a theory of mathematical truth. A foundation for problem solving must specify a goal-oriented formalism in which problems can be stated. Furthermore there must be a formalism for specifying the allowable methods of solution. As part of the definition of the formalisms, the following elements must be defined: the data structure, the control structure, and the primitive procedures. Being a theory of actions, a foundation for problem solving must confront the problem of change: How can account be taken of the changing situation in the world? In order for there to be problem solving, there must be an active agent called a problem solver. A foundation for problem solving must consider how much knowledge and what kind of knowledge problem solvers can have about themselves. In contrast to the foundation of mathematics, the semantics for a foundation for problem solving should be defined in terms of properties of procedures. We would like to see mathematical investigations on the adequacy of the foundations for problem solving provided by PLANNER. In chapter 8 we have begun one kind of such an investigation.

To be more specific, a foundation for problem solving must concern itself with the following complex of topics:

PROCEDURAL EMBEDDING: How can "real world" knowledge be effectively embedded in procedures. What are good ways to express problem solution methods and how can plans for the solution of problems be formulated?

GENERALIZED COMPILATION: What are good methods for transforming high level goal-oriented language into efficient algorithms.

VERIFICATION: How can it be verified that a procedure does what is intended.

PROCEDURAL ABSTRACTION: What are good methods for abstracting general procedures from special cases.

One formulation of a foundation for problem solving requires that there should be two distinct formalisms:

1: A METHODS formalism which specifies the allowable methods of solution

2: A PROBLEM SPECIFICATION formalism in which to pose problems.

The problem solver is expected to figure out how to combine its available methods in order to produce a solution which satisfies the problem specification. One of the aims of the above formulation of problem solving is to clearly separate the methods of solution from the problems posed so that it is impossible to "cheat" and give the problem solver the methods for solving the problem along with the statement of the problem. We propose to bridge the chasm between the methods formalism and the problem formalism. Consider more carefully the two extremes in the specification of processing:

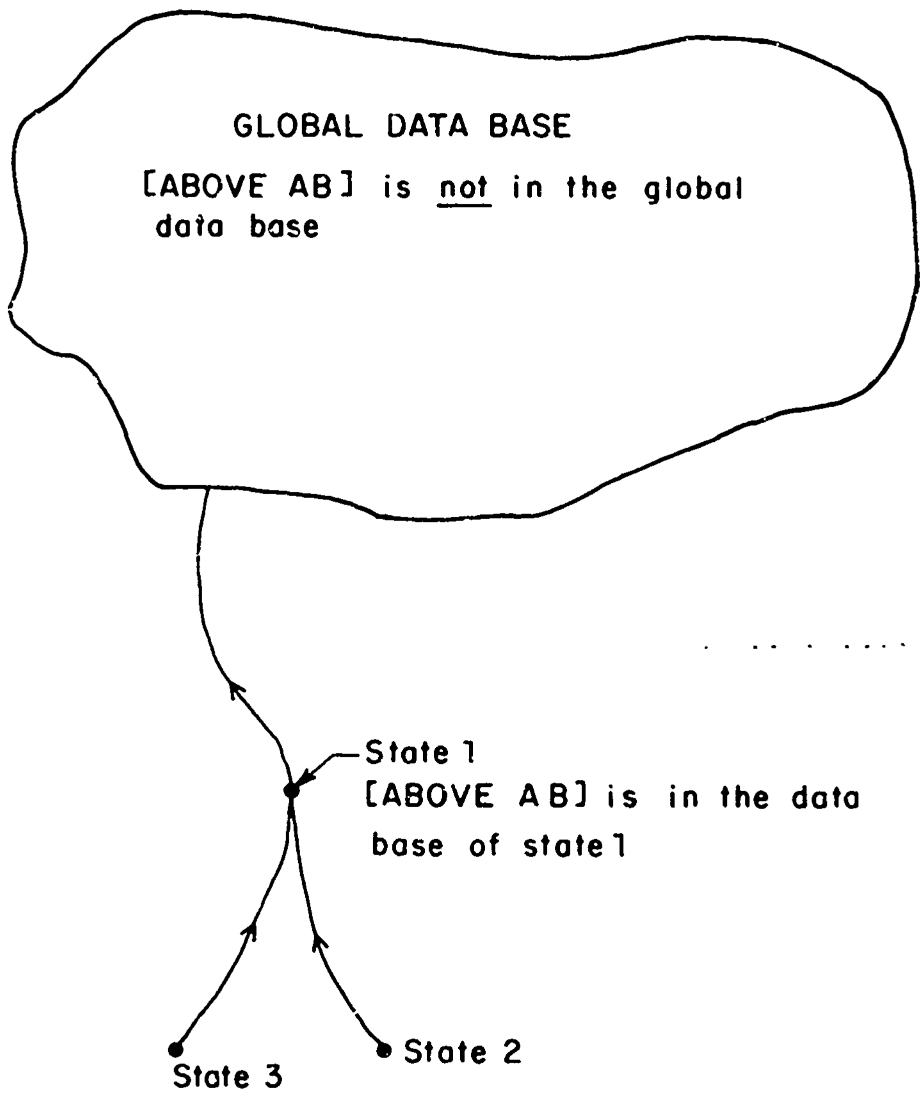
A: Explicit processing (e.g. methods) is the ability to specify and control actions down to the finest details.

B: Implicit processing (e.g. problems) is the ability to specify the end result desired and not to say much about how it should be achieved.

PLANNER attempts to provide a formalism in which a problem solver can bridge the continuum between explicit and implicit processing. We aim for a maximum of flexibility so that whatever knowledge is available can be incorporated, even if it is fragmentary and heuristic.

PLANNER is a high level, goal-oriented formalism in which one can specify to a large degree what one wants done rather than how to do it. Many of the primitives in PLANNER are concerned with manipulating a data base in a pattern directed fashion. Most of the primitives have been developed as extensions to the formalism when we have found problems that could not otherwise be solved in a natural way. Of course the trick is to incorporate the new primitive as a genuine extension of wide applicability. Others have suggested themselves as adjuncts in order to obtain useful closure properties in the formalism. We would be grateful to any reader who could suggest problems that would seem to require further extensions or modifications to the formalism.

There are many ways in which one can approach a description of PLANNER. In this section we will describe PLANNER from an Information Processing Viewpoint. To do this we will describe the data structure and the control structure of the formalism.


GLOBAL DATA BASE
[ABOVE AB] is not in the global data base
State 1
[ABOVE AB] is in the data base of state 1
State 3
State 2

PLANNER ALLOWS FOR THE SIMULTANEOUS EXISTENCE OF INCOMPATIBLE LOCAL STATES IN MODELS.

DATA STRUCTURE:

GRAPH MEMORY forms the basis for PLANNER's data space which consists of directed graphs with labeled arcs. The operation of PUTTING and GETTING the components of data objects have been generalized to apply to any data type whatsoever. For example to PUT the value CANONICAL on the expression <+ X Y <* X Z>> under the indicator SIMPLIFIED is one way to record that <+ X Y <* X Z>> has been canonically simplified. Then the degree to which an expression is simplified can be determined by GETTING the value under the indicator SIMPLIFIED of the expression. The operations of PUT and GET can be implemented efficiently using hash coding. Lists and vectors have been introduced to gain more efficiency for common special purpose structures. The graph memory is useful to PLANNER in many ways. Monitoring gives PLANNER the capability of trapping all read, write, and execute references to a particular data object. The monitor (which is found under the indicator MONITOR) of the data object can then take any action that it sees fit in order to handle the situation. The graph memory can be used to retrieve the value of an identifier i of a process p by GETTING the i component of p. Code can be commented by simply PUTTING the actual comment under the indicator COMMENT. Also graph memory enables unique copies of structures to be efficiently and conveniently stored.

DATA BASE: What is most distinctive about the way in which PLANNER uses data is that it has a data base in which data can be inserted and removed. For example inserting [AT B1 P2] into the data base might signify that block B1 is at the place P2. A coordinate of an expression is defined to be an atom in some position. An expression is determined by its coordinates. Assertions are stored in buckets by their coordinates using the graph memory in order to provide efficient retrieval. In addition a total ordering is imposed on the assertions so that the buckets can be sorted. Imperatives as well as declaratives can be stored in the data base. We might assert that whenever an expression of the form [AT object1 place1] is removed from the data base, then any expression in the data base of the form [ON object1 object2] should also be removed from the data base. The data base can be tree structured so that it is possible to simultaneously have several local data bases which are incompatible. Furthermore assertions in the data base can have varying scopes so that some will last the duration of a process while others are temporary to a subroutine.

CONTROL STRUCTURE: PLANNER uses a pattern directed multiprocess backtrack control structure to tie the operation of its primitives together.

BACKTRACKING: PLANNER processes have the capability of backtracking to previous states. A process can backtrack into a procedure activation (i.e. a specific instance of an invocation of a procedure) which has already returned with a result. Using the theory of comparative schematology, we have proved in chapter 8 that the use of backtrack control enables us to achieve effects that a language (such as LISP) which is limited to recursive control cannot achieve. Backtracking preserves the nesting of the subroutine structure of PLANNER while allowing the consequences of elaborate tentative hypotheses to be explored without losing the capability of rejecting the hypotheses and all of their consequences. A choice can be made on the basis of the available knowledge and if it doesn't work, a better choice can be made using the new information discovered while investigating the first choice. Also backtrack control makes PLANNER procedures easier to debug since they can be run backwards as well as forwards enabling a problem solver to "zero in" on bugs.

MULTIPROCESSING gives PLANNER the capability of having more than one locus of control in problem solving. By using multiple processes, arbitrary patterns of investigation through a conceptual problem space can be carried out. Processes can have the power to create, read, write, interrupt, resume, single step, and fork other processes. The ability to single-step or to interrupt processes allows the definition of procedures which are NOT monotone in the sense of lattice theory. Potentially the failure of monotonicity is a serious flaw in the lattice theoretic approach towards a mathematical foundation for effective procedures.

PATTERN DIRECTION combines aspects of control and data structure. The fundamental principle of pattern directed computation is that a procedure should be a pattern of what the procedure is intended to accomplish. In other words a procedure should not only do the right thing but it should appear to do the right thing as well! PLANNER uses pattern direction for the following operations:

CONSTRUCTION of structured data objects is accomplished by templates. We can construct a list whose first element is the value of x and whose second element is the value of y by the procedure (x y). If x has the value 3 and y has the value (A B) then (x y) will evaluate to (3 (A B)).

DECOMPOSITION is accomplished by matching the data object against a structured pattern. If the pattern (x1 x2) is matched against the data object ((3 4) A) then x1 will be given the value (3 4) and x2 will be given the value A.

RETRIEVAL: An assertion is retrieved from the data base by specifying a pattern which the assertion must match and thereby bind the identifiers in the pattern. For example we can determine if there is anything in the data base of the form [ON x A]. If [ON B A] is the only item in the data base, then x is bound to B. If there is more than one item in the data base which matches a retrieval pattern, then an arbitrary choice is made. The fact that a choice was made is remembered so that if a simple failure backtracks to the decision, another choice can be made.

INVOCATION: Procedures can be invoked by patterns of what they are supposed to accomplish. Suppose that we have a stopped sink. One way we could try to solve the problem would be to know the name of a plumber whom we could call. An alternative which is more analogous to pattern directed invocation is to advertise the fact that we have a stopped sink and the qualifications needed to fix it. In PLANNER this is accomplished by making the advertisement (i.e. a pattern which represents what is desired) into a goal. The procedure invoked by the pattern might or might not succeed in achieving the goal depending on the environment in which it was called. The procedure invoked can be required to undo all the actions that it took to try to achieve the goal. For example if we were unhappy with the way in which a plumber fixed our sink, we could require that he restore the situation to its previous state. Since many theorems might match a goal, a recommendation is allowed as to which of the candidate theorems might be useful. The recommendation is a pattern which a candidate theorem must match.

One basic idea behind PLANNER is to exploit the duality that we find between certain imperative and declarative sentences. Consider the statement (implies A B). The statement is a perfectly good declarative. In addition, it can also have certain imperative uses for PLANNER. It can say that we might set up a procedure which will note whether A is ever asserted and if so to consider the wisdom of asserting B in turn. [Note: it is not always wise! Suppose we assert <integer 0> and (implies <integer n> <integer (+ n 1)>)].

Furthermore it permits us to set up a procedure that will watch to see if it is ever our goal to try to deduce B and if so whether A should be made a subgoal. Exactly the same observations can be made about the contrapositive of the statement {implies A B} which is {implies {not B} {not A}}. Statements with universal quantifiers, conjunctions, disjunctions, etc. can also have both declarative and imperative uses. PLANNER theorems are used as imperatives when executed and as declaratives when used as data. The imperative analogues have the advantage that they can more easily express any procedural knowledge that we might have such as "Don't use this theorem twice".

Our work on PLANNER has been an investigation in PROCEDURAL EPISTEMOLOGY, the study of how knowledge can be embedded in procedures. The THESIS OF PROCEDURAL EMBEDDING is that intellectual structures should be analyzed through their PROCEDURAL ANALOGUES. We will try to show what we mean through examples:

DESCRIPTIONS are procedures which recognize how well some candidate fits the description.

PATTERNS are descriptions which match configurations of data. For example <either 4 <atomic>> is a procedure which will recognize something which is either 4 or is atomic.

DATA TYPES are patterns used in declarations of the allowable range and domain of procedures and identifiers. More generally, data types have analogues in the form of procedures which create, destroy, recognize, and transform data.

GRAMMARS: The PROGRAMMAR language of Terry Winograd another step towards one kind of procedural analogue for natural language grammar.

SCHEMATIC DRAWINGS have as their procedural analogue methods for recognizing when particular figures fit within the schemata.

PROOFS correspond to plans for recognizing and expanding valid chains of deductions. Indeed many proofs can fruitfully be considered to define procedures which are proved to have certain properties. For example a proof by mathematical induction of a effective formula p[n] can be considered to be a proof that the following function always returns "TRUE":

```
p[n] := if p[0] then "TRUE" else p[n-1]
```

Conversely, proofs by execution induction of properties of procedures can be used to demonstrate mathematical facts. For example proofs by execution induction can imitate proofs by mathematical induction:

```
        <f n> := <repeat out [[i 0]]
                         ;"initialize i to 0"
                         Intent:  p[i]
                         <cond
                            [<is? .i .n>
                                ;"if .i is equal to .n then
                                          exit with the value
.n"
                                <.out .n>]>
                         <_ :i <+ .i 1>>
                         ;"else increment i and repeat">
```

Proving the intention p[i] by execution induction will establish that for all n we have p[n]. Proofs by execution induction enable global properties (such as convergence and equivalence) to be proved by purely local analysis.

MODELS are collections of procedures for simulating the behavior of the system being modeled. MODELS of PROGRAMS are procedures for defining properties of procedures and attempting to verify the properties so defined. Models of programs can be defined by procedures which state the relations that must hold as control passes through the program.

PLANS are general, goal oriented procedures for attempting to carry out some task.

THEOREMS of the QUANTIFICATIONAL CALCULUS have as their analogues procedures for carrying out the deductions which are justified by the theorems. For example, consider a theorem of the form {IMPLIES x y}. One procedural analogue of the theorem is to consider whether x should be made a subgoal in order to try to prove something of the form y.

DRAWINGS: The procedural analogue of a drawing is a procedure for making the drawing. Rather sophisticated display processors have been constructed for making drawings on cathode ray tubes.

RECOMMENDATIONS: PLANNER has primitives which allow recommendations as to how disparate sections of goal oriented language should be linked together in order to accomplish some particular task.

GOAL TREES are represented by a snapshot of the instantaneous configuration of problem solving processes.

One corollary of the thesis of procedural embedding is that learning entails the learning of the procedures in which the knowledge to be learned is embedded. Another aspect of the thesis of procedural embedding is that the process of going from general goal oriented language which is capable of accomplishing some task to a special purpose, efficient, algorithms especially designed for the task should itself be mechanized. By expressing the properties of the special purpose algorithm in terms of their procedural analogues, we can use the analogues to establish that the special purpose routine does in fact do what it is intended.

From the above observations, we have constructed a formalism that permits both the imperative and declarative aspects of statements to be easily manipulated. PLANNER uses a pattern-directed information retrieval system. The data base is interrogated by specifying a pattern of what is to be retrieved. Instead of having to explicitly name procedures which are to be called, they can be invoked implicitly by a pattern (this important concept is called PATTERN-DIRECTED INVOCATION). When a statement is asserted, recommendations determine what conclusions will be drawn from the assertion. Procedures can make recommendations as to which theorems should be used in trying to draw conclusions from an assertion, and they can recommend the order in which the theorems should be applied. Goals can be created and automatically dismissed when they are satisfied. Objects can be found from schematic or partial descriptions. Provision is made for the fact that statements that were once true in a model may no longer be true at some later time and that consequences must be drawn from the fact that the state of the model has changed. Assertions and goals created within a procedure can be dynamically protected against interference from other procedures. Unlike some other formalisms such as GPS, PLANNER has no explicit goal tree. Instead the computation itself can be thought to be investigating some conceptual problem space. Primitives for a multiprocess backtrack control structure give flexibility to the ways in which the conceptual problem space can be investigated. Procedures written in the formalism are extendable in that they can make use of new knowledge whether it be primarily declarative or imperative in nature. Hypotheses can be established and later discharged. PLANNER has been used to write a block control language in which we specify how blocks can be moved around by a robot. We would like to write a structure building formalism in which we could provide descriptions of structures (such as houses and bridges) and let PLANNER figure out how to build them. The logical deductive system used by PLANNER is subordinate to the hierarchical control structure of the language. PLANNER theorems operate within a context consisting of return addresses, goals, assertions, bindings, and local changes of state that have been made to the global data base. Through the use of this context we can guide the computation and avoid doing basically the same work over and over again. For example, once we determine that we are working within a group (in the mathematical sense) we can restrict our attention to theorems for working on groups since we have direct control over what theorems will be used. PLANNER has a sophisticated deductive system in order to give us greater power over the direction of the computation. Of course procedures written in PLANNER are not intrinsically efficient. A great deal of thought and effort must be put into writing efficient procedures. PLANNER does provide some basic mechanisms and primitives in which to express problem solving procedures. The control structure can still be used when we limit ourselves to using resolution as the sole rule of inference. A uniform proof procedure gives very little control over how or when a theorem is used. The problem is one of the level of the interpreter that is used. A digital computer by itself will only interpret the hardware instructions of the machine. A higher level interpeter such as LISP will interpret assignments and recursive function calls. At a still higher level an interpreter such as MATCHLESS will interpret patterns for constructing and decomposing structured data. PLANNER can interpret assertions, find statements, and goals. It goes without saying that code can be compiled for any of the higher level interpeters so that it actually runs under a lower level interpreter. In general higher level interpreters have greater choice in the actions that they can take since instructions are phrased more in terms of goals to be achieved rather than in terms of explicit elementary actions. The problem that we face is to raise the level of the interpreter while at the same time keeping the actions taken by it under control. Due to its extreme hierarchical control and its ability to make use of new imperative as well as declarative knowledge, it is feasible to carry out very long chains of inference in PLANNER without extreme inefficiency.

We are concerned as to how a theorem prover can unify structural problem solving methods with domain dependent algorithms and data into a coherent problem solving process. By structural methods we mean those that are concerned with the formal structure of the argument rather than with the semantics of its domain dependent content.

An example of a structural method is the "consequences of the consequent" heuristic. By the CONSEQUENCES OF THE CONSEQUENT heuristic, we mean that a problem solver should look at the consequences of the goal that is being attempted in order to get an idea of some of the statements that could be useful in establishing or rejecting the goal.

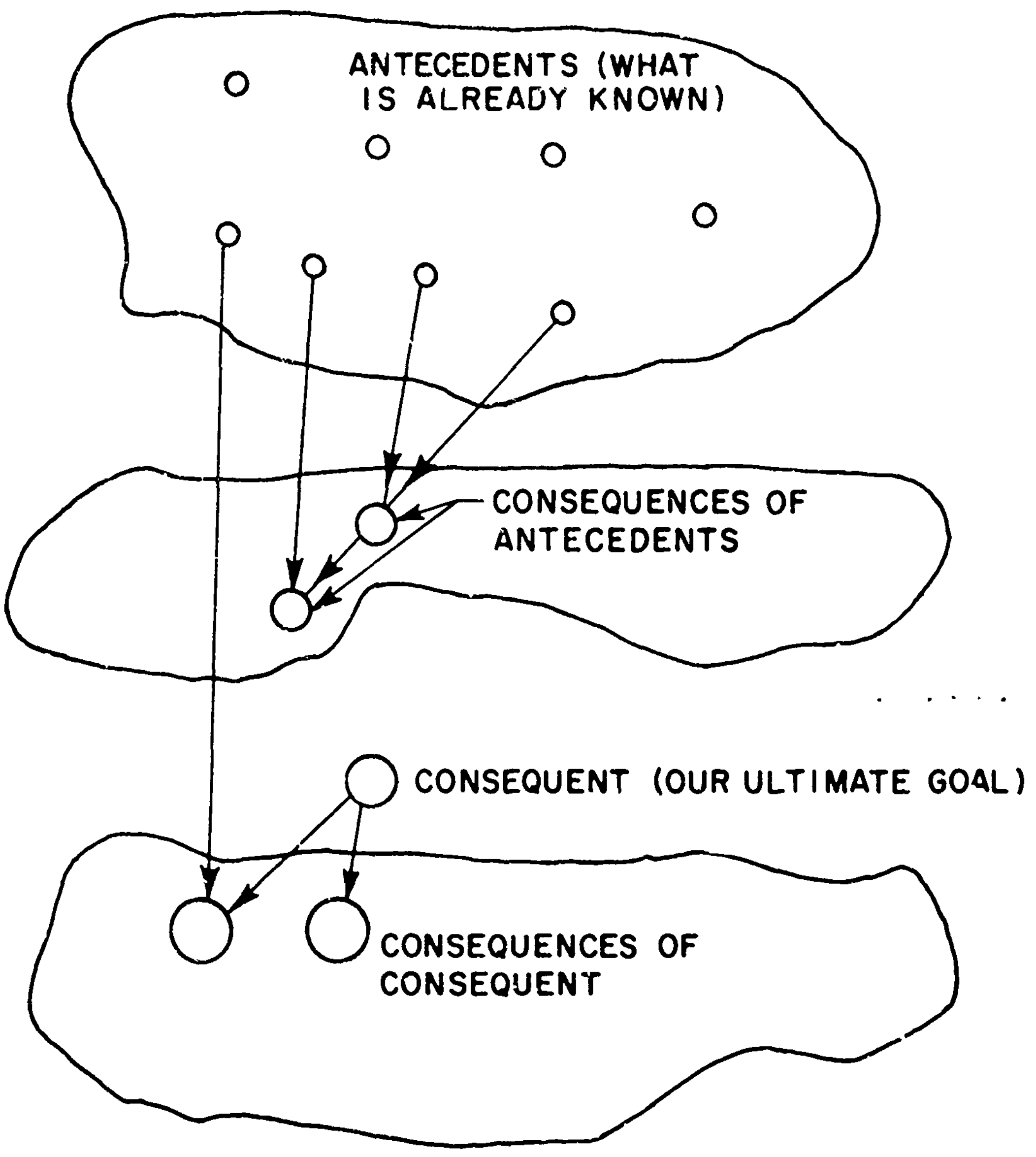
We need to discover more powerful structural methods. PLANNER is intended to provide a computational basis for expressing structural methods. One of the most important ideas in PLANNER is that it brings some of the structural methods of problem solving out into the open where they can be analyzed and generalized. There are a few basic patterns of looping and recursion that are in constant use among programmers. Examples are recursion on binary trees as in LISP and the FIND statement of PLANNER. The primitive FIND will construct a list of the objects with certain properties. For example we can find five things which are on something which is green by evaluating

```
        <FIND 5 x
                <GOAL [ON x y]>
                <GOAL [GREEN y]>>
```

which reads "find 5 x's such that x is ON y and y is GREEN.

The patterns of looping and recursion represent common structural methods used in programs. They specify how commands can be repeated iteratively and recursively. One of the main problems in getting computers to write programs is how to use these structural patterns with the particular domain dependent commands that are available. It is difficult to decide which if any of the basic patterns is appropriate in any given problem. The problem of synthesizing programs out of canned loops is formally identical to the problem of finding proofs using mathematical induction. We have approached the problem of constructing procedures out of goal oriented language from two directions. The first is to use canned loops (such as the FIND statement) where we assume a-priori the kind of control structure that is needed. The second approach is to try to abstract the procedure from protocols of its action in particular cases.

2·page 32a

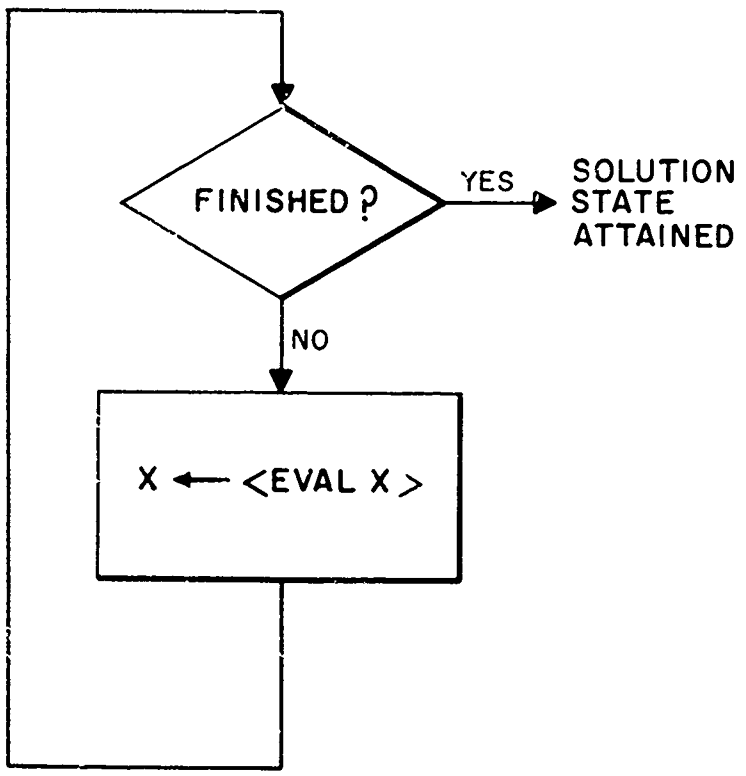

ANTECEDENTS (WHAT IS ALREADY KNOWN)
CONSEQUENCES OF ANTECEDENTS
CONSEQUENT (OUR ULTIMATE GOAL)
CONSEQUENCES OF CONSEQUENT

Another structural method is PROGRESSIVE REFINEMENT. The way problems are solved by progressive refinement is by repeated evaluation. Instead of trying to do a complete investigation of the problem space all at once, repeated refinements are made. For example in a game like chess the same part of the game tree might be looked at several times. Each time certain paths are more deeply explored in the light of what other investigations have revealed to be the key features of the position. Problems in design seem to be particularly suitable for the use of progressive refinement since proposed designs are often amenable to successive refinement. The way in which progressive refinement typically is done in PLANNER is by repeated evaluation. Thus the expression which is evaluated to solve the problem will itself produce as its value an expression to be evaluated.

The task of artificial intelligence is to program inanimate machines to perform tasks that require intelligence. Over the past decade several different approaches toward A. I. have developed. Although very pure forms of these approaches will seldom be met in practice, we find that it is useful for purposes of discussion to consider these conceptual extremes. One approach (called results mode by S. Papert) has been to choose some specific intellectual task that humans can perform with facility and write a program to perform it. Several very fine programs have been written following this approach. One of the first was the Logic Theorist which attempted to prove theorems in the propositional calculus using the deductive system

3 page 33a

PROGRESSIVE REFINEMENT



developed in Principia Mathematica. The importance of the Logic Theorist is that it developed a body of techniques which when cleaned up and generalized have proved to be fundamental to furthering our understanding of A. I. The results mode approach offers the potentiality of maximum efficiency in solving particular classes of problems. On the other hand, there have been a number of programs written from the results mode approach which have not advanced our understanding although the programs achieved slightly better results than had been achieved before. These programs have been large, clumsy, brute force pieces of machinery. There is a clear danger that the results mode approach can degenerate into trying to achieve A. I. via the "hairy kludge a month plan". The problems with "hairy kludges" are well known. It is impossible to get such programs to communicate with each other in a natural and intimate way. They are difficult to understand, extend, and modify because of the ad hoc way in which they are constructed.

Another approach to A. I. that has been prominent in the last decade is that of the uniform proof procedure. Proponents of the approach write programs which accept declarative descriptions of combinatorial problems and then attempt to solve them. In its most pure form the approach does not permit the machine to be given any information as to how it might solve its problems. The character table approach to A. I. is a modification of the uniform procedure approach in which the program is also given a finite state table of connections between goals and methods. The uniform procedure approach offers a great deal of elegance and a maximum of a certain kind of generality. Current programs that implement the uniform procedure approach suffer from extreme inefficiency. We believe that the inefficiency is intrinsic in the approach.

PLANNER is not neccessarily general in the same sense that a uniform proof procedure is general. PLANNER is intended to be a natural computational basis for methods of solving problems in a domain. A complete proof procedure for a quantificational calculus is general in the sense that if one can force the problem into the form of the input language and is prepared to wait eons if necessary, then the computer is guaranteed to find a solution if there is one. The approach taken in PLANNER is to subordinate the deductive system to an elaborate hierarchical control structure. Although PLANNER itself is domain independent, procedures written in it have differing overlapping degrees of domain independence. Proponents of the uniform procedure approach are apt to say that PLANNER "cheats" because through the use of its hierarchical control structure, it is possible to tell the program how to try to solve its problems. In order to prevent this kind of "cheating", they would restrict the input to consist entirely of declaratives. But surely, it is to the credit of a program that it is able to accept new imperative information and make use of it. A problem solver needs a high level language for expressing problem solving methods even if the language is only used by the problem solver to express its problem solving methods to itself. PLANNER serves both as the language in which problems are posed to the problem solver and the language in which methods of solution are formulated. PLANNER is not intended to be a solution to the problem of finding general methods for reducing the combinatorial search involved to test whether a given proposition is valid or not. It is intended to be a general formalism in which knowledge of a domain can be combined and integrated. Realistic problem solving programs will need vast amounts of knowledge. We consider all methods of solving problems to be legitimate. If a program should happen to already know the answer to the problem that it is asked to solve, then it is perfectly reasonable for the problem to be solved by table look-up. We should use the criterion that the problem solving power of a program should increase much faster than in direct proportion to the number of things that it is told. The important factors in judging a program are its power, elegance, generality, and efficiency.

3. Discursive Overview

This chapter contains an explanation of some of the ideas in PLANNER in essay form. It is partially based on a draft written by T. Winograd for the course 6.545. If the reader would like to see a more systematic presentation, he can consult the subsequent chapters.

The easiest way to understand PLANNER is to watch how it works, so in this section we will present a few simple examples and explain the use of some of its most elementary features. These examples are not intended to represent TOY PROBLEMS to serve as test cases for "general problem solvers". The toy problem paradigm is misleading because toy problems can be solved without any real knowledge of the domain in which the toy problem is posed. Indeed, it seems gauche to use any thing as powerful as real knowledge on such simple problems. In contrast we believe that real world problems require vast amounts of procedural knowledge for their solution. We see it as part of our task to provide the intellectual capabilities needed for effective problem solving. We would like to see the toy problem paradigm replaced with an INTELLECTUAL CAPABILITY paradigm where the object is to illustrate the intellectual capabilities needed so that knowledge can be effectively embedded in procedures.

First we will take the most venerable of traditional deductions:

```
    Turing is a human
    All humans are fallible
 so
    Turing is fallible.
```

It is easy enough to see how this could be expressed in the usual logical notation and handled by a uniform proof procedure. Instead, let us express it in one possible way to PLANNER by saying:

```
<ASSERT [HUMAN TURING]>

<ASSERT <DEFINE THEOREM1
        <CONSEQUENT [Y] [FALLIBLE ?Y]
                <GOAL [HUMAN ?Y]>>>>
```

Function calls are enclosed between "<" and ">". The proof would be generated by asking PLANNER to evaluate the expression:

```
<GOAL [FALLIBLE TURING]>
```

The example illustrates several points about PLANNER. First, there are at least two different kinds of information stored in the data base: declaratives and imperatives. Notice that for complex sentences containing quantifiers or logical connectives we have a choice whether to express the sentence by declaratives or by imperatives.

Second, one of the most important points about PLANNER is that it is an evaluator for statements. It accepts input in the form of expressions written in the PLANNER language and evaluates them, producing a value and side effects. ASSERT is a function which, when evaluated, stores its argument in the data base of assertions. In this example we have defined a theorem of the CONSEQUENT type [we will see other types later]. This states that if we ever want to establish a goal of the form [FALLIBLE ?Y], we can do this by accomplishing the goal [HUMAN ?Y], where Y is an identifier. The strange prefix character "?" is part of PLANNER's pattern matching capabilities [which are extensive and make use of the pattern-matching language MATCHLESS which is explained in chapter 4 of the dissertation]. If we ask PLANNER to prove a goal of the form [A Y], there is no obvious way of knowing whether A and Y are constants [like TURING and HUMAN in the example] or identifiers. LISP solves this problem by using the function QUOTE to indicate constants. In pattern matching this is inconvenient and makes most patterns much bulkier and more difficult to read. Instead, PLANNER uses the opposite convention -- a constant is represented by the atom itself, while an identifier must be indicated by adding an appropriate prefix. This prefix differs according to the exact use of the identifier in the pattern, but for the time being let us just accept "?" as a prefix indicating an identifier. The definition of the theorem indicates that it has one identifier, Y by the [Y] following CONSEQUENT.

The third statement illustrates the function GOAL, which tries to prove an assertion. This can function in several ways. If we had asked PLANNER to evaluate <GOAL [HUMAN TURING]> it would have found the requested assertion immediately in the data base and succeeded [returning as its value some indicator that it had succeeded]. However, [FALLIBLE TURING] has not been asserted, so we must resort to theorems to prove it. Later we will see that a GOAL statement can give PLANNER various kinds of advice on which theorems are applicable to the goal and should be tried. For the moment, take the default case, in which the evaluator tries all theorems whose consequent is of a form which matches the goal [i.e. a theorem with a consequent [?Z TURING] would be tried, but one of the form [HAPPY ?Z] or [FALLIBLE ?Y ?Z] would not]. Assertions can have an arbitrary list structure for their format -- they are not limited to two-member lists or three-member lists as in these examples. The theorem we have just defined would be found, and in trying it, the match of the consequence to the goal would cause the identifier Y to be bound to the constant TURING. Therefore, the theorem sets up a new goal [HUMAN TURING] and this succeeds immediately since it is in the data base. In general, the success of a theorem will depend on evaluating a PLANNER program of arbitrary complexity. In this case it contains only a single GOAL statement, so its success causes the entire theorem to succeed, and the goal [FALLIBLE TURING] is proved. The following is the protocol of the evaluation:

```
<GOAL [FALLIBLE TURING]> [FALLIBLE TURING] is not in the data base
so attempt to invoke a theorem to esablish the goal
    enter THEOREM1
    Y becomes TURING
    <GOAL [HUMAN TURING]> is satisfied since the goal is in the
    data base
return [FALLIBLE TURING]
```

The way in which identifiers are bound by matching is of key importance to PLANNER. Consider the question "Is anything fallible?", or in logic [EXISTS X [FALLIBLE X]]. This could be expressed in

PLANNER as:

```
     <PROG [X] <GOAL [FALLIBLE ?X]>>
```

Notice that PROG [PLANNER's equivalent of a LISP PROG] in this case acts as an existential quantifier. It provides a binding-place for the identifier X, but does not initialize it -- it leaves it in a state particularly marked as unassigned. To answer the question, we ask PLANNER to evaluate the entire PROG expression above. To do this it starts by evaluating the GOAL expression. This searches the data base for an assertion of the form [FALLIBLE ?X] and fails. It then looks for a theorem with a consequent of that form, and finds the theorem we defined above. Now when the theorem is called, the identifier Y in the theorem is linked to the identifier X in the goal, but since X has no value yet, Y does not receive a value. The theorem then sets up the goal [HUMAN ?Y] with Y as an identifier. The PLANNER primitive GOAL uses the data-base retrieval mechanism to look for any assertion which matches that pattern [i.e. an instantiation], and finds the assertion [HUMAN TURING]. This causes Y [and therefore X] to be bound to the constant TURING, and the theorem succeeds, completing the proof and returning the value [FALLIBLE TURING].

There seems to be something missing. So far, the data base has contained only the relevant objects, and therefore PLANNER has found the right assertions immediately. Consider the problem we would get if we added new information by evaluating the statements:

```
        <ASSERT [HUMAN SOCRATES]>
        <ASSERT [GREEK SOCRATES]>
```

Our data base now contains the assertions:

```
[HUMAN TURING]
[HUMAN SOCRATES]
[GREEK SOCRATES]
```

and theorem1:

```
<CONSEQUENT [Y] [FALLIBLE ?Y]
        <GOAL [HUMAN ?Y]>>
```

What if we now ask, "Is there a fallible Greek?" In PLANNER we would do this by evaluating the expression:

```
<PROG [X]
        <GOAL [FALLIBLE ?X]>
        <GOAL [GREEK ?X]>>
```

If PLANNER runs into a failure trying to evaluate an expression, then it backtracks to the last decision that was made and dumps the responsibility of how to proceed on the procedure which made the decision. Notice what might happen. The first GOAL may be satisfied by exactly the same deduction as before, since we have not removed information. If the data-base retriever happens to run into TURING before it finds SOCRATES, the goal [HUMAN ?Y] will succeed, binding Y and thus X to TURING. After [FALLIBLE ?X] succeeds, the PROG will then establish the new goal [GREEK TURING], which is doomed to fail since it has not been asserted, and there are no applicable theorems. If we think in LISP terms, this is a serious problem, since the evaluation of the first GOAL has been completed before the second one is called, and the "stack" now contains only the return address for PROG and the identifier X. If we try to go back to the beginning and start over, it will again find TURING and so on, ad infinitum.

One of the most important features of the PLANNER language is that backtracking in case of failure is always possible, and moreover this backtracking can go to the last place where a decision of any sort was made. Here, the decision was to pick a particular assertion from the data base to match a goal. Another kind of decision is the choice of a theorem to try to achieve a goal. PLANNER keeps enough information to change any decision and send evaluation back down a new path.

In our example the decision was made inside the theorem for FALLIBLE, when the goal [HUMAN ?Y] was matched to the assertion [HUMAN TURING]. PLANNER will retrace its steps, try to find a different assertion which matches the goal, find [HUMAN SOCRATES], and continue with the proof. The theorem will succeed with the value [FALLIBLE SOCRATES], and the PROG will proceed to the next expression, <GOAL [GREEK ?X]>. Since X has been bound to SOCRATES, this will set up the goal [GREEK SOCRATES] which will succeed immediately by finding the corresponding assertion in the data base. Since there are no more expressions in the PROG, it will succeed, returning as its value the value of the last expression, [GREEK SOCRATES]. The whole course of the deduction process depends on the failure mechanism for backtracking and trying things over [this is actually the process of trying different branches down the conceptual goal tree.] This then is the PLANNER executive which establishes and manipulates subgoals in looking for a proof.

We would now like to give a somewhat more formal description of the behavior of PLANNER on the above problem. If we intoduce suitable notation our problem solving protoccls can be made much more succinct and their structure made visible. Also by formalizing the notions, we can make PLANNER construct and analyze protocols. This provides one kind of tool by which PLANNER can understand its own behavior and make generalizations on how to proceed.

In this case the protocol is:

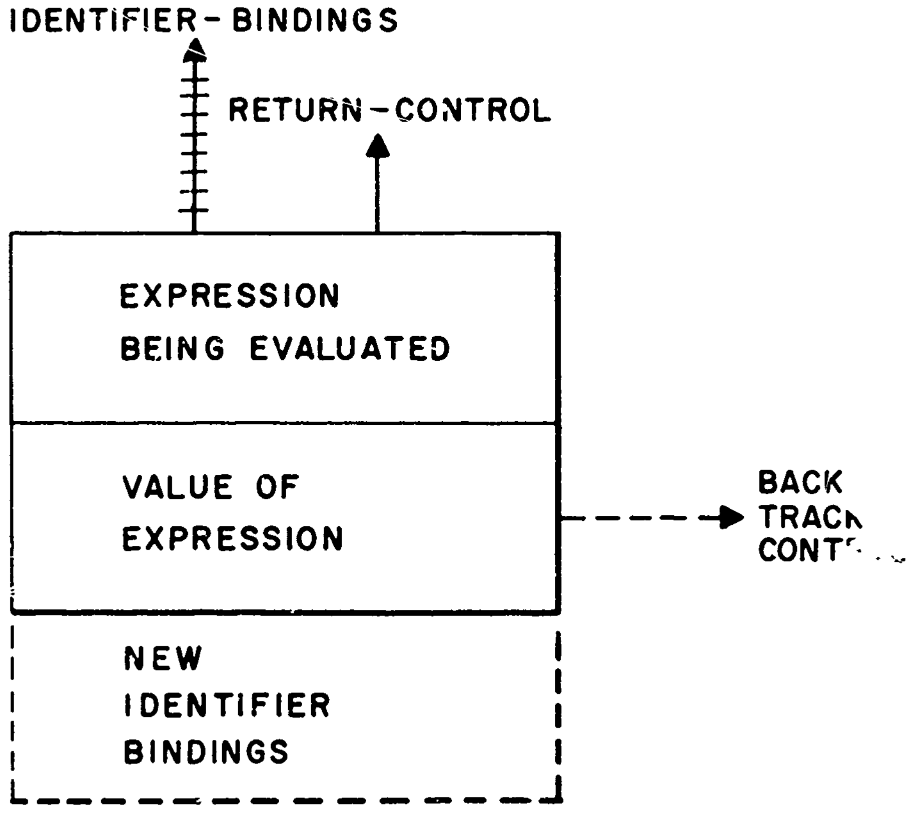
```
1:  enter PROG
    2:  X is rebound but not initialized
    3:  <GOAL [FALLIBLE ?X]> will attempt a pattern directed
    invocation since nothing in the data base matches [FALLIBLE ?X].
        4:  enter THEOREM1
        5:  match [FALLIBLE ?Y] with [FALLIBLE ?X] thus linkin-
        the situation is shown in snapshot number 1
            6:  <GOAL [HUMAN ?Y]> finds [HUMAN TURING] in the da
            base
                7:  Y gets the value TURING thus giving X the va
                TURING
            8:  return [HUMAN TURING]
        9:  THEOREM1 returns [FALLIBLE TURING]
    10:  <GOAL [GREEK TURING]> fails since it is not in the data base
    and there are no matching consequents
```

Thus PLANNER must backtrack to step 7 and try again. The situation is shown in snapshot number 2. For the convenience of the reader, we will repeat the first six steps from above and then continue the protocol.

```
1:  enter PROG
    2:  X is rebound but not initialized
    3:  <GOAL [FALLIBLE ?X]>
        4:  enter THEOREM1
        5:  match [FALLIBLE ?Y] with [FALLIBLE ?X] thus linking Y to X
            6':  <GOAL [HUMAN ?Y]> finds [HUMAN SOCRATES] in the data
            base
                11:  Y gets the value SOCRATES thus giving X the value
                SOCRATES
            12:  return [FALLIBLE SOCRATES]
```

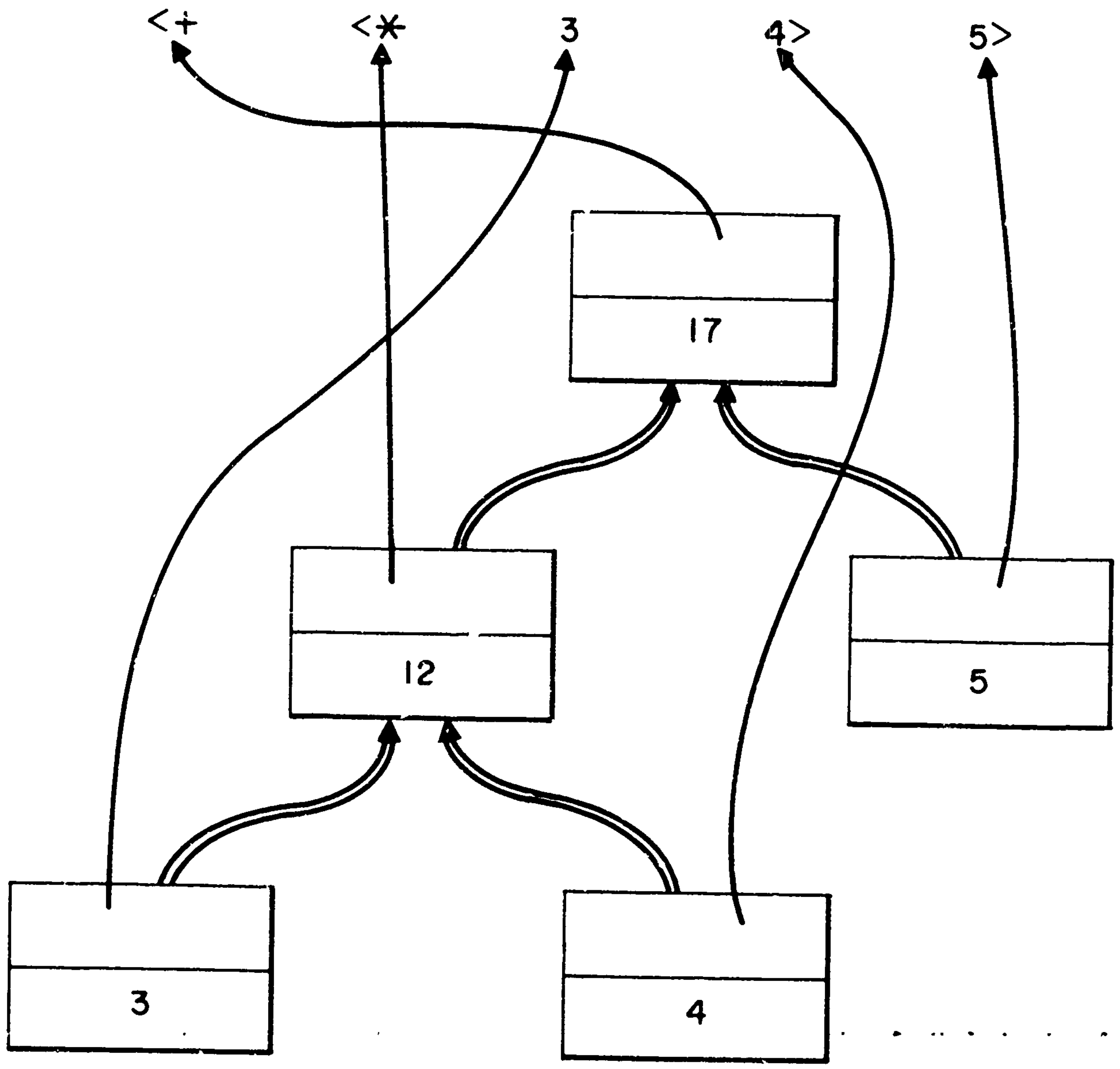
3. page 44a

FORMAT OF FUNCTION ACTIVATIONS
IN SNAPSHOTS



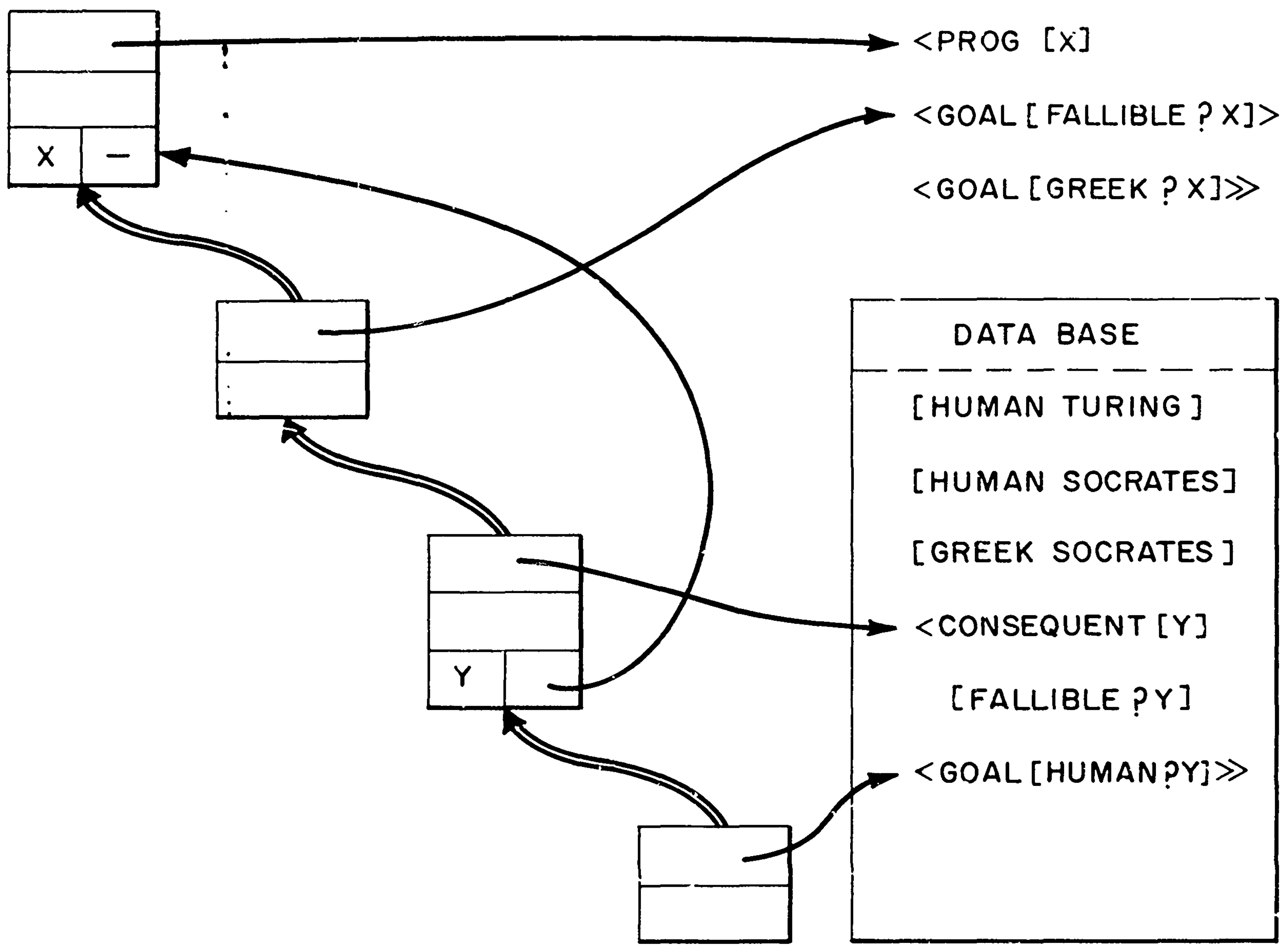
NOTE: THE IDENTIFIER-BINDINGS AND RETURN-CONTROL POINTERS OF AN ACTIVATION ARE USUALLY THE SAME AND THUS ARE COMBINED INTO A DOUBLE POINTER LIKE THIS $\Longrightarrow$

SNAPSHOT OF EVALUATION OF

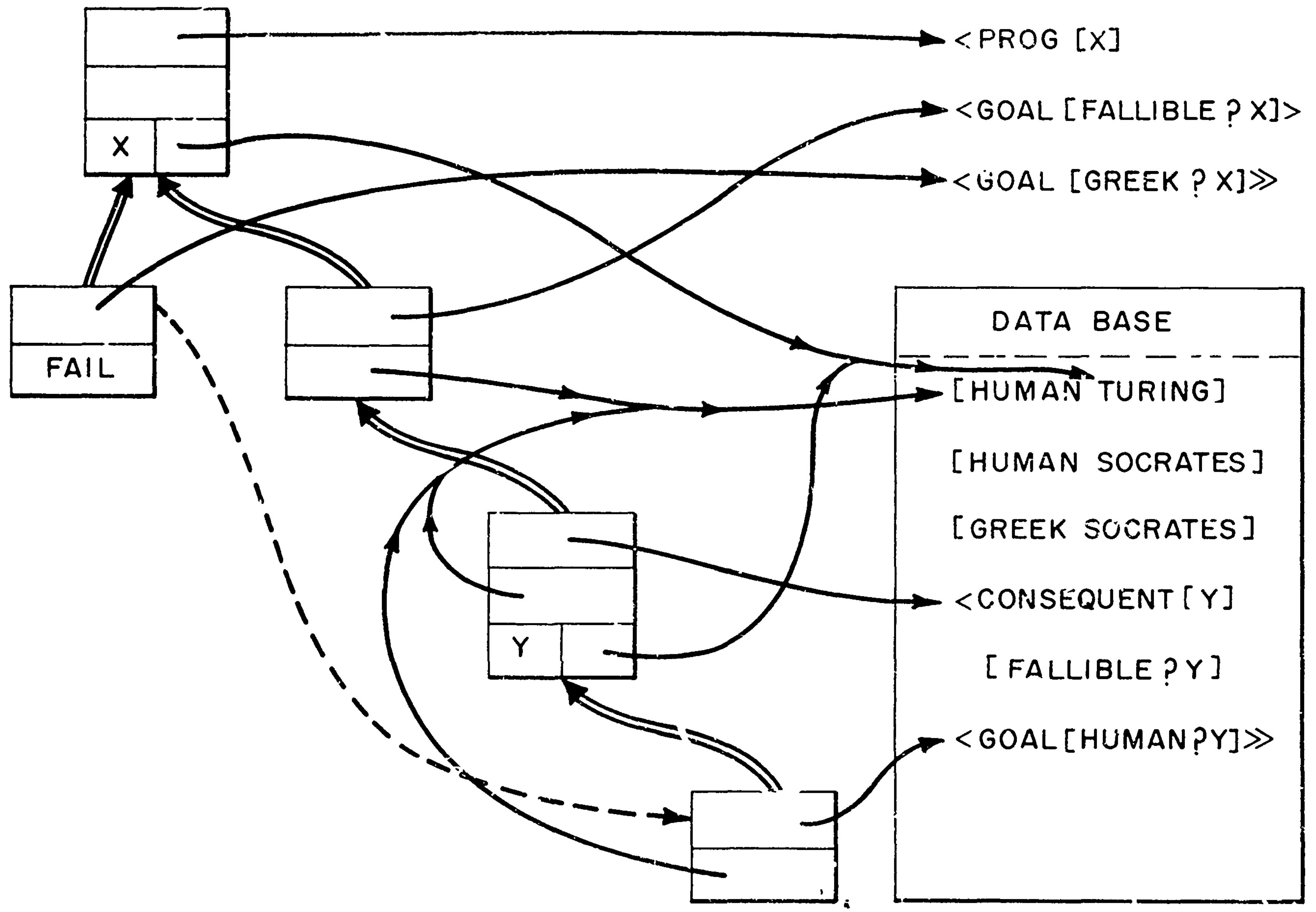


THUS <+ <* 3 4> 5> EVALUATES TO 17.

SNAPSHOT NO. 1



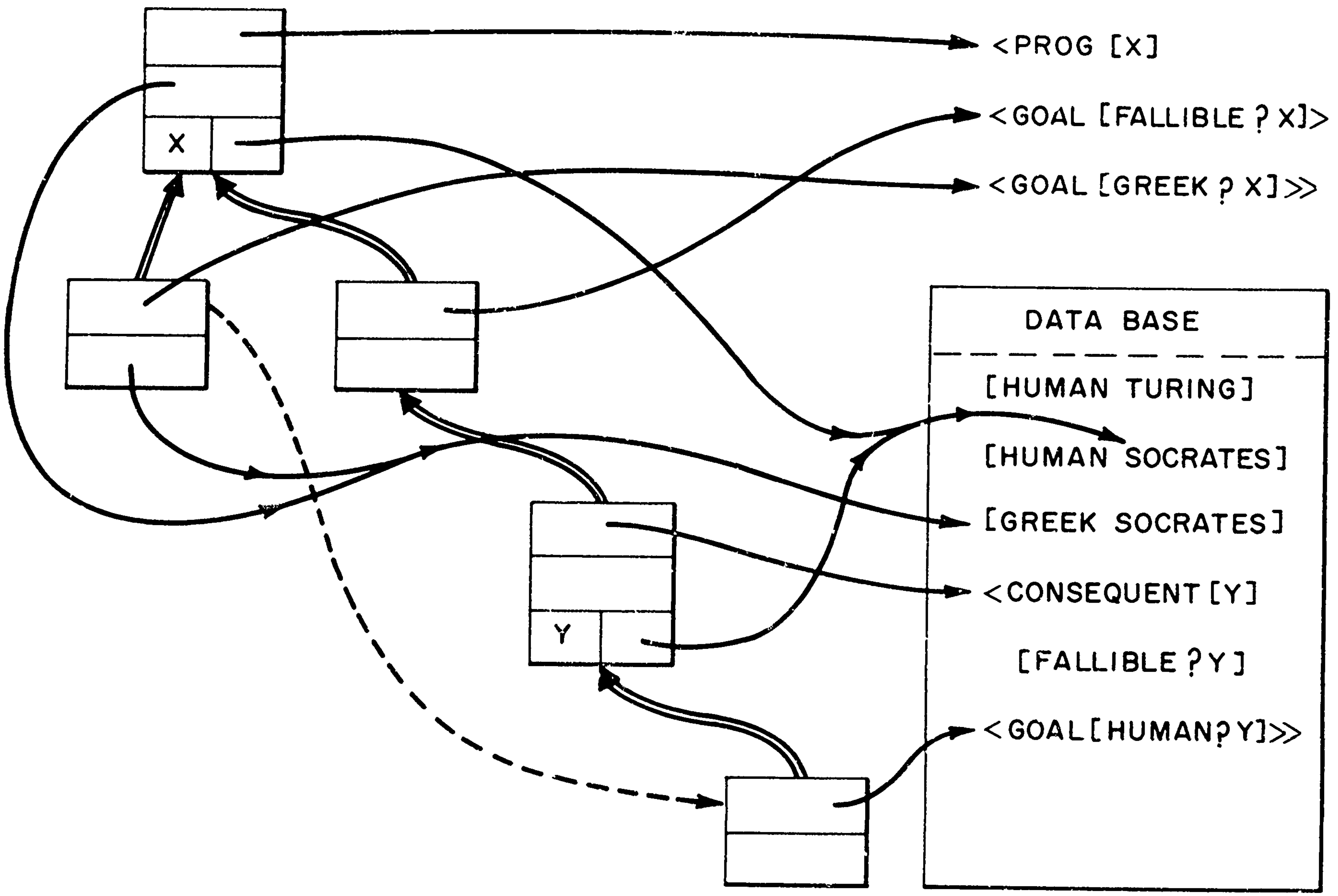
SNAPSHOT NO. 2



3. page 44d

3. pag 44e

SNAPSHOT NO. 3



```
        13:  THEOREM1 returns [FALLIBLE SOCRATES]
    14:  <GOAL [GREEK SOCRATES]>
15:  return [GREEK SOCRATES] as the top level value
```

The situation is shown in snapshot number 3.

So far we have seen that although PLANNER is written as an evaluator, it differs in several critical ways from anything which is normally considered a programming language. First, it is goal-directed. Theorems can be thought of as subroutines, but they can be called by specifying the goal which is to be satisfied. This is like having the abilitiy to say "Call a subroutine which will achieve the desired result at this point." Second, the evaluator has the mechanism of success and failure to handle the exploration of the conceptual goal tree. In PLANNER there is no explicit goal tree. The conceptual goal tree is represented by a SNAPSHOT of a CONFIGURATION of PROCESSES. Thus PLANNER has powerful control structure primitives to allow the conceptional goal structure to be easily and naturally reflected in the execution of PLANNER processes. Other evaluators, such as LISP, with a basic recursive evaluator have no way to do this. One of our current areas of research is to increase the richness of the machinery provided by PLANNER to guide the movement to the goal. Third, PLANNER contains a large set of primitive commands for matching patterns and manipulating a data base, and for handling that data base efficiently.

On the other side, we can ask how it differs from other theorem provers. What is gained by writing theorems in the form of programs, and giving them power to call other programs which manipulate data? The key is in the form of the data the theorem-prover can accept. Most systems take declarative information, as in predicate calculus. This is in the form of expressions which represent "facts" about the world. These are manipulated by the theorem-prover according to some fixed uniform process set by the system. PLANNER can make use of imperative information, telling it how to go about proving a subgoal, or to make use of an assertion. This produces what is called HIERARCHICAL control structure. That is, any theorem can indicate what the theorem prover is supposed to do as it continues the proof. It has the full power to evaluate expressions which can depend on both the data base and the subgoal tree, and to use its results to control the further proof by making assertions, deciding what theorems are to be used, and specifying a sequence of steps to be followed. What does this mean in practical terms? In what way does it make a "better" theorem prover? We will give several examples of areas where the approach is important.

First, consider the basic problem of deciding what subgoals to try in attempting to satisfy a goal. Very often, knowledge of the subject matter will tell us that certain methods are very likely to succeed, others may be useful if certain other conditions are present, while others may be possibly valuable, but not likely. We would like to have the ability to use heuristic programs to determine these facts and direct the theorem prover accordingly. It should be able to direct the search for goals and solutions in the best way possible, and be able to bring as much intelligence as possible to bear on the decision. In PLANNER this is done by adding to our GOAL statement a recommendation list which can specify that ONLY certain theorems are to be tried, or that certain ones are to be tried FIRST in a specified order. Since theorems are programs, subroutines of any type can be called to help make this decision before establishing a new GOAL. Each theorem has a name [in our definition on page 1, the theorem was given the name THEOREM1], to facilitate referring to them explicitly.

Another important problem is that of maintaining a data base with a reasonable amount of material. Consider the first example above. The statement that all humans are fallible, while unambiguous in a declarative sense is actually ambiguous in its imperative sense [i.e. the way it is to be used by the theorem prover]. The first way is to simply use it whenever we are faced with the need to prove [FALLIBLE ?X]. Another way might be to watch for a statement of the form [HUMAN ?X] to be asserted, and to immediately assert [FALLIBLE ?X] as well. There is no abstract logical difference, but the impact on the data base is tremendous. The more conclusions we draw when information is asserted, the easier proofs will be, since they will not have to make the additional steps to deduce these consequences over and over again. However since we don't have infinite speed and size, it is clearly folly to think of deducing and asserting everything possible [or even everything interesting] about the data when it is entered. If we were working with totally abstract meaningless theorems and axioms [an assumption which would not be incompatible with many theorem-proving schemes], this would be an insoluble dilemma. But PLANNER is designed to work in the real world, where our knowledge is much more structured than a set of axioms and rules of inference. We may very well, when we assert [LIKES ?X POETRY] want to deduce and assert [HUMAN ?X], since in deducing things about an object, it will very often be relevant whether that object is human, and we shouldn't need to deduce it each time. On the other hand, it would be silly to assert [HAS-AS-PART ?X SPLEEN], since there is a horde of facts equally important and equally limited in use. Part of the knowledge which PLANNER should have of a subject, then, is what facts are important, and when to draw consequences of an assertion. This is done by having theorems of an antecedent type:

```
<ASSERT <DEFINE THEOREM2
    <ANTECEDENT [X Y] [LIKES ?X ?Y]
            <ASSERT [HUMAN ?X]>>>>
```

This says that when we assert that X likes something, we should also assert [HUMAN ?X]. Of course, such theorems do not have to be so simple. A fully general PLANNER program can be activated by an ANTECEDENT theorem, doing an arbitrary [that is, the programmer whether he be man or machine has free choice] amount of deduction, assertion, etc. Knowledge of what we are doing in a particular problem may indicate that it is sometimes a good idea to do this kind of deduction, and other times not. As with the CONSEQUENT theorems, PLANNER has the full capacity when something is asserted, to evaluate the current state of the data and proof, and specifically decide which ANTECEDENT theorems should be called.

PLANNER therefore allows deductions to use all sorts of knowledge about the subject matter which go far beyond the set of axioms and basic deductive rules. PLANNER itself is subject-independent, but its power is such that the deduction process never needs to operate on such a level of ignorance. The programmer can put in as much heuristic knowledge as he wants to about the subject, just as a good teacher would help a class to understand a mathematical theory, rather than just telling them the axioms and then giving theorems to prove.

Another advantage in representing knowledge in an imperative form is the use of a theorem prover in dealing with processes involving a sequence of events. Consider the case of a robot manipulating blocks on a table. It might have data of the form, "block1 is on block2," "block2 is behind block3", and "if x is on y and you put it on z, then x is on z, and is no longer on y unless y is the same as z". Many examples in papers on theorem provers are of this form [for example the classic "monkey and bananas" problem]. The problem is that a declarative theorem prover cannot accept a statement like [ON B1 B2] at face value. It clearly is not an axiom of the system, since its validity will change as the process goes on. It usually is put in a form [ON B1 B2 S0] where S0 is a symbol for an initial state of the world. The third statement might be expressed as:

```
[FOR-ALL TOPBLOCK NEWSUPPORT OLDSUPPORT S
        [AND
                [ON TOPBLOCK NEWSUPPORT [PUT TOPBLOCK NEWSUPPORT S]]
                [OR
                        [EQUAL NEWSUPPORT OLDSUPPORT]
```

```
                    [NOT [CN
                            TOPBLOCK
                            OLDSUPPORT
                            [PUT TCPBLOCK NEWSUPPORT S]]]]]]
```

In this representaticn, [PUT X Y S] is the state which results from putting X on Y when the previous state was S. We run into a problem when we try to ask [CN Z W [PUT X Y S]] i.e. is block Z on block W after we put X on Y? A human knows that if we haven't touched Z or W we could just ask [ON Z W S] but in general it may take a complex deduction to decide whether we have actually moved them, and even if we haven't, it will take a whole chain of deductions [tracing back through the time sequence] to prove they haven't been moved. In PLANNER, where we specify a process directly, this whole type of problem can be handled in an intuitively more satisfactory way by using the primitive function ERASE.

Evaluating <ERASE [CN ?X ?Y]> removes the assertion [ON ?X ?Y] from the data base. If we think of theorem provers as working with a set of axioms, it seems strange to have function whose purpose is to erase axioms. If instead we think of the data base as the "state of the world" and the operation of the prover as manipulating that state, it allows us tc make great simplificaticns. Now we can simply assert [ON B1 B2] without any explicit mention of states. We can express the necessary theorem as:

```
 <ASSERT <DEFINE THEOREM3
        <CCNSEQUENT [TCPBLOCK NEWSUPPCRT OLDSUPPORT]
            [PUT  ?TOPELOCK ?NEWSUPPORT]
                <GOAL [ON ?TOPBLOCK ?CLDSUPPCRT]>
                <ERASE [ON ?TOPBLOCK ?CLDSUPPORT]>
                <ASSERT [ON ?TCPBLOCK ?NEWSUPPORT]>>>>
```

This says that whenever we want to satisfy a goal of the form [PUT ?TOPBLOCK ?NEWSUPPORT], we should first find out what thing OLDSUPPORT the thing TOPBLOCK is sitting on, erase the fact that it is sitting on OLDSUPPORT, and assert that it is sitting on NEWSUPPORT. We could also do a number of other things, such as proving that it is indeed possible to put TOPBLOCK on NEWSUPPORT, or adding a list of specific instructions to a movement plan for an arm to actually execute the goal. In a more complex case, other interactions might be involved. For example, if we are keeping assertions of the form [ABOVE ?X ?Y] we would need to delete those assertions which became false when we erased [ON ?X ?Z] and add those which became true when we added [ON ?X ?Y]. ANTECEDENT theorems would be called by the assertion [ON ?X ?Y] to take care of that part, and a similar group called ERASING theorems can be called in an exactly analogous way when an assertion is erased, to derive consequences of the erasure. Again we emphasize that which of such theorems would be called is dependent on the way the data base is structured, and is determined by knowledge of the subject matter. In this example, we would have to decide whether it was worth adding all of the ABOVE relations to the data base, with the resultant need to check them whenever something is moved, or instead to omit them and take time to deduce them from the ON relation each time they are needed.

Thus in PLANNER, the changing state of the world can be mirrored in the changing state of the data base, avoiding any need to make explicit mention of states, with the requisite overhead of deductions. This is possible since the information is given in an imperative form, specifying theorems as a series of specific steps to be executed. PLANNER also allows the construction of local data bases called states which are variants of the global data base. Evaluation of PLANNER expressions is carried out relative to a local state. Thus simultaneous consideration can be given to two incompatible states of the world by explicitly calling the evaluator to evaluate statements in the two states.

If we look back to the distinction between assertions and theorems made at the beginning of this chapter, it would seem that we have established that the base of assertions is the "current state of the world", while the base of theorems is our permanent knowledge of how to deduce things from that state. This is not exactly true, and one of the most exciting possibilities in PLANNER is the capability for the program itself to create and modify the PLANNER functions which make up the theorem base. Rather than simply making assertions, a particular PLANNER function might be written to put together a new theorem or make changes to an existing theorem, in a way dependent on the data and current knowledge. It seems likely that meaningful "teaching" involves this type of behavior rather than simply modifying parameters or adding more individual facts [assertions] to a declarative data base.

For example suppose we are given the following protocols for a function f. An expression such as "new [5 * 4]" means that we are introducing a new identifier which is 5 * 4 = 20.

```
<f C> : 0=0 IS TRUE SO 1
Thus <f 0> = 1
```

The above expression reads, "to compute <f 0> you test 0=0 which is true sc the answer is 1".

```
<f 1> : 1=0 IS FALSE SO
       1 * new [1-1] 0=0 IS TRUE SO 1
Thus <f 1> = 1
```

The above expression reads, "to compute <f 1> you test 1=0 which is false so the answer is 1 times the quantity which is computed by first computing the intermediate result 1-1 then testing if 0=0 which is true so the quantity is 1."

```
<f 2> : 2=0 IS FALSE SO
       2 * new [2-1] 1=0 IS FALSE SO
               1 * new [1-1] 0=0 IS TRUE SO 1
Thus <f 2> = 2 * 1 * 1 = 2

<f 3> : 3=0 IS FALSE SO
       3 * new [3-1] 2=0 IS FALSE SO
           2 * new [2-1] 1=0 IS FALSE SC
               1 * new [1-1] 0=0 IS TRUE SO 1
Thus <f 3> = 3 * 2 * 1 * 1 = 6
```

By the process of "variab lization", we conclude that the above protocols are compatible with the following program which is in the form of a tree [which we shall call the protocol tree].

```
<f x0> = if xC=0 then 1
       else x0 * new [[x0-1]=x1] if x1=0 then 1
               else x1 * new [[x1-1]=x2] if x2=0 then 1
                       else x2 * new [[x2-1]=x3]
                               if x3=0 then 1
                               else...
```

Now by identifying indistinguishable nodes on the protocol tree, we obtain:

```
       <f x> = if x=0 then 1
               else x *<f [x-1]>
```

The reader will note that f is the factorial function. PLANNER procedures and theorems can be taught in precisely the same fashion [which we call procedural abstraction]. For example, the computer can be taught to build a wall or recognize a tower from examples. The reader is cautioned that although we shall speak of the computer being "taught", we do not assume that anything like what has been classically described as "learning" is taking place. We assume that the teacher has a good working model of the student that is being taught and that he honestly attempts to convey a certain body of knowledge to the student. Of course the student will be told anything which might help him to understand the material faster.

Procedural abstraction is one way in which a special purpose routine can be constructed from general goal oriented language. We would like to express the intended properties of the special purpose routine so that we can establish that the routine really does what it is supposed to do. For example we might be interested in establishing that the function divide defined below satisfies its intentions.

```
<define divide <function idivide
       ;"let idivide be name of this activation"
       [n d]
       ;"the function divide is a function of two arguments n and d"
       <repeat [[r .n] [q 0]]
              ;"initialize r to n and q to zero"
              ;"we are in a repeat loop which will repeatedly
                     execute the following expressions"
              <cond
                 [<is? <less .d> .r>
                     ;"if .r is less than .d then"
                     <.idivide .q .r>
                     ;"exit the activation named
                            idivide with .q and .r"]>
              <assign :r <- .r .d>>
```

```
                ;"assign r the value of r minus d"
                <assign :q <+ .q 1>>
                ;"assign q the value of q plus 1"
                ;"now go back and do the body of the repeat
                      loop all over again">>>
```

We shall express the intentions of the function DIVIDE in a goal oriented formalism called INTENDER. INTENDER enables us to embed the intentions for a program in the text of the program. The easiest way to understand INTENDER is to watch how it works. In order to show how it works we must first define some intentions. INTENDER introduces two new primitives OVERALL and INTENT to express intentions in code. The primtive OVERALL expresses the overall intention of a function or loop whereas INTENT asserts that the intended situation really holds within the body of the function or loop. The meaning of the intentions embedded in the function DIVIDE are explained below. INTENDER is a giant sledge hammer to use to squash such a tiny problem. The reader can see this sledge hammer used on harder problems in chapter 7. INTENDER needs to be able to talk about function calls in a pattern directed way. We will use !' to suppress procedural invocations. Thus whereas <+ 3 5> evaluates to the NUMBER 8, the expression !'<+ 3 5> will evaluate to the CALL <+ 3 5>. Assertions which contain calls constitute a still higher level assertion than the two which we have introduced thus far. The semantics of [illegible] assertions are determined in part by the body of the procedure wh[illegible] led. For example the assertion that !'<= !'<+ 1 2> !'<+ 2 1>> can be established from the DEFINITICN of +. Similarly in a very incestuous way, we can make assertions about PLANNER procedures whose intentions are themselves written in PLANNER and at any given time constitute the model that PLANNER has of itself! By using intentions expressed in PLANNER, there is nothing that in principle PLANNER cannot be made to understand about itself.

```
<define divide <function idivide [n d]
<overall [ ]
    <intention [ ]
        <and
                <goal !'<is !'<greater 0> .n>>
                <goal !'<is !'<greater 0> .d>>>
        <and
                <assert !'<is !'<greater 0> .n>>
                <assert !'<is !'<greater 0> .d>>>>
    <repeat [[r .n] [q 0]]
        !;<intention
            <goal !'<= .n !'<+ .r !'<* .d .q>>>>
            <assert !'<= .n !'<+ .r !'<* .d .q>>>>>
        <cond
            [<is? <less .d> .r>
                <.idivide .q .r>]>
        <assign :r <- .r .d>>
        <assign :q <+ .q 1>>>
    <function [Q R]
        <intention [ ]
                <and
                        <assert !'<= .n !'<+ .R !'<* .d .Q>>>>
                        <assert !'<is? !'<less .d> .R>>>
                <and
                        <goal !'<= .n !'<+ .R !'<* .d .Q>>>>
                        <goal !'<is? !'<less .d> .R>>>>>>>>>
```

The overall intention for the function DIVIDE is that it return two values Q and R which we assert will have the property that

```
        !'<=.n !'<+ .R !'<* .d .Q>>>
```

The inside intent of the function DIVIDE is the goal that DIVIDE will return two values Q and R which will have the property that

```
        !'<=.n !'<+ .R !'<* .d .Q>>>
```

The body of DIVIDE is a REPEAT loop with two locals r and q which are respectively initialized to 0 and n. The overall intention of the REPEAT loop is the goal

```
        !'<= .n !'<+ .r !'<* .d .q>>>
```

The REPEAT loop has an intent that asserts that

```
        !'<=.n !'<+ .r !'<* .d .q>>>
```

at the top of the loop.

The intentions for DIVIDE are proved by running them in INTENDER. The intentions are verified abstractly. Thus they must be true independent of what the actual arguments to the function are. We shall use the notation x_n for the nth value of the identifier x with x_ being an abbreviation for the initial value of x. The actions of INTENDER on the intentions of DIVIDE are as follows:

```
        From the overall all intentiion of the function we have:
        <assert !'<is !'<greater 0> n_>>
        <assert !'<is !'<greater 0> d_>>

        The following assertions come from the declarations of the
repeat loop
        <assert '<= r_ n_>>
        <assert '<= q_ 0>>

        The intention of the repeat statement on first entry is
satisfied:
        <goal !'<=
                n_
                !'<+
                        r_
                        !'<* d_ q_>>>>

        We inductively assume for the repeat loop
        <assert !'<=
                n_
                !'<+
                r_1
                !'<* d_ q_1>>>>

enter intentions of COND
There are two cases for the conditional:

Case1:
<assert !'<is?
                !'<less d_>
                r_1>>

        From the overall intention we have:
        Q becomes q_1
        R becocomes r_1
        <goal !'<=
                n_
                !'<+
                        r_1
                        !'<* d_ q_1>>>>
        <goal !'<is? !'<less d_> r_1>>

Case2:
<assert !'<is?
                !'<greater= d_>
                r_1>>

        From <assign :r <- .r .d>> we get:
        <assert !'<=
                        r_2
                        !'<- r_1 d_>>>

        From <assign :q <+ .q 1>> we get:
        <assert !'<=
                        q_2
                        !'<+ q_1 1>>>

        The recursive goal is satisfied by simplification:
        <goal !'<=
                n_
                !'<+
                        r_2
                        !'<* d_ q_2>>>>
```

4. THE PATTERN MATCHING LANGUAGE MATCHLESS

MATCHLESS is a pattern directed language that is used in the implementation of PLANNER. MATCHLESS is used both in the internal workings of PLANNER and as a tool in the deductive system itself. MATCHLESS is similar in certain respects to other structural pattern matching languages such as CONVERT and SNOBOL. It has been designed with the following considerations in mind:

0. The language must obey the Fundamental Principle of Pattern Directed Computation: the procedure body should be a pattern that describes the purpose of the procedure. The principle has been developed even further in PLANNER where procedures are invoked on the basis of their intent.

1. The language should be very powerful yet simple constructs should be efficiently compiled. By incorporating more knowledge into a program, it must be possible to increase its efficiency up to the limits imposed by the machine on which it runs.

2. Functions must be able to be separately compiled.

3. It should not require parsing for efficient interpretation. Procedures should be naturally and efficiently constructed and edited by other procedures.

4. The language must interface with PLANNER in a natural way since it is used as a basic part of the deductive system. Effective

problem solving requires a sophisticated programmable matcher.

5. The language should treat strings, lists, vectors, tuples, and nodes symmetrically so that for the most part the same program will run whether the structures are made up of vectors, tuples, nodes, or lists. Declarations determine which form is actually used.

6. The language should have no automatic coercion. Any procedures which wish to coerce their arguments should be able to do so easily.

7. The language should have only one mode of evaluation for value. Locatives should always be generated explicitly in the same way.

8. All the loops of the language should be guaranteed to be properly nested.

4.1 The Syntax of Identifiers and Expressions

MATCHLESS attempts to obey the Fundamental Principle of Pattern Directed Computation: the procedure body should be a pattern of what the procedure is supposed to accomplish. For example it allows the list (a b c) to be produced by simply evaluating (a b c). In attempting to realize the principle we have been led to develop a certain amount of syntax which (unfortunately!) must be described.

4.1.1 Prefix Operators for Identifiers

As is usual in pattern matching languages we shall allow constants like 3, a, (a b), and (e (f g)) to match only themselves. An identifier is indicated by a prefix operator which tells how the identifier is to be used. For example .x is the element value of the identifier x. If x has the value (a 3) then .x will only tch (a 3). We need to be able to change the value of an identifier in a pattern match. Suppose that x has the value 3. If we match _x [the tentative value of x] against (a b), then x is given the value (a b). The identifier x will keep the value (a b) if the remainder of the pattern matches. Otherwise the value of x will revert to 3. Again suppose that x has the value 3. If we match :x [the altered value of x] against (a b), then x is given the value (a b). However the value of x will remain (a b) whether or not the remainder of the pattern matches.

The above prefix operators are actually defined in terms of procedure calls. We are not enamored with the syntax of the prefix operators but they are easier to type than the procedures listed below.

A small meta syntax is needed in order to give explanations of the primitives of the language. We shall use | to delimit metasyntatic variables which are elements and - to delimit those which are sequences.

The following table explains the prefix operators which yield element values:

.|x| = <VALUE |x|> the element value of the identifier |x|

,|x| = <GLOBAL |x|> the element global value of |x|

The following table explains the prefix operators which match elements:

?|x| = <GIVEN |x|> will give |x| the value of the matching element if |x| does not already have a value; otherwise ?|x| will only match the value of |x|.

:|x| = <ALTER!-PERSISTENT |x|> will alter the value of x to be the matching element even if |x| already has a value.

_|x| = <ALTER!-TENTATIVE |x|> will tentatively alter the value of |x| to be the matching element but if a failure backs up then the old value of |x| will be restored.

If x has the value (a 1) then (b .x 4) will evaluate to (b (a 1) 4). The character ! is the esca character. We will use !.x to denote the segment value of the identifier x. For example (b !.x 4) will evaluate to (b a 1 4). In each case preceding the prefix operator for an identifier will result in the segment prefix operator for that identifier. If we match the pattern (c !:x d) against the value (c 3 a d) then x will be given (3 a) as its value.

The following table explains the prefix operators which yield segment values:

!.|x| = (VALUE |x|) the segment value of the identifier |x|

!,|x| = (GLOBAL |x|) the segment global value of |x|

The following table explains the prefix operators which match segments:

!?|x| = (GIVEN x) will give x the value of the matching segment if x does not already have a value; otherwise !?x will only match the value of x.

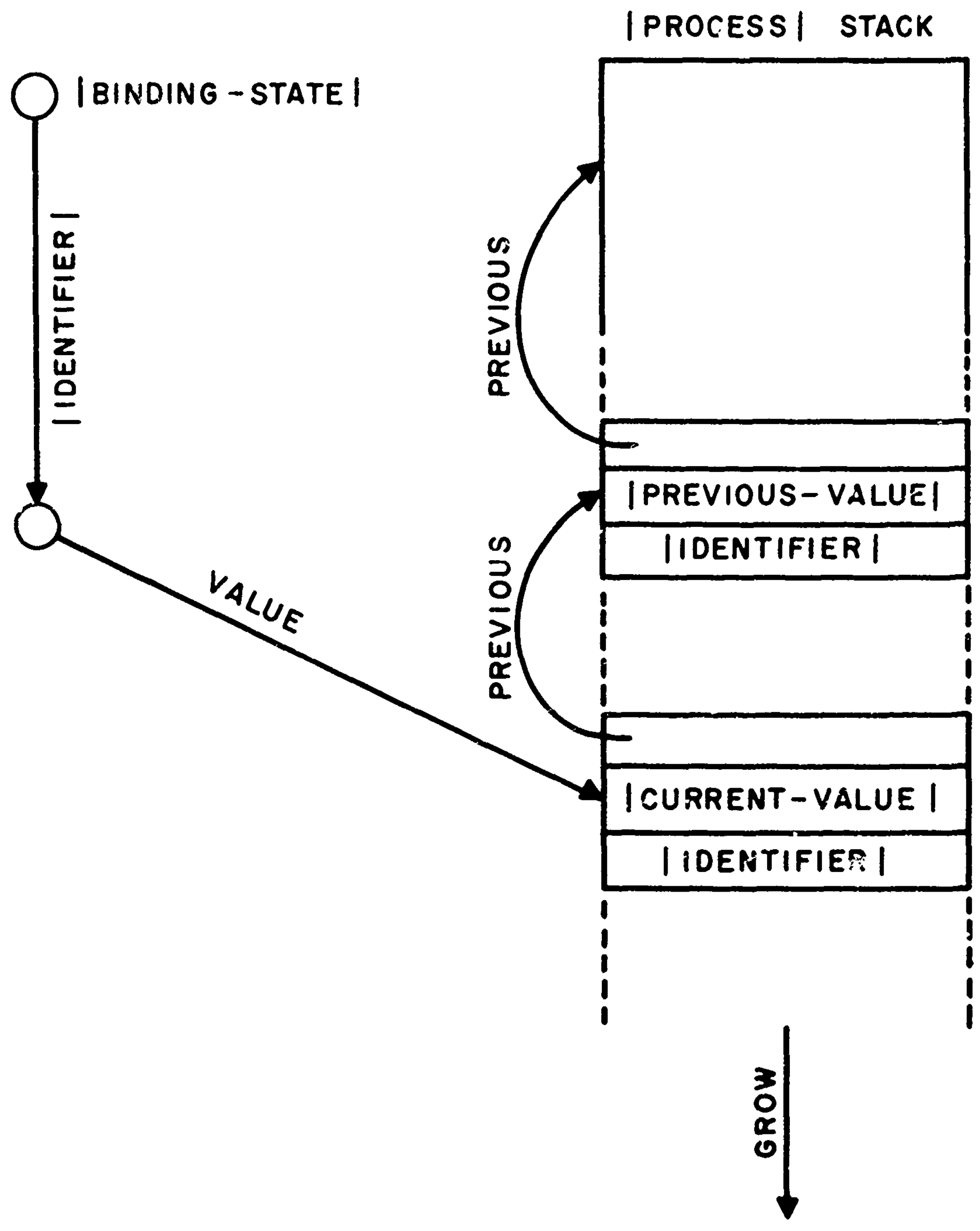
!:x = (ALTER!-PERSISTENT x) will alter the value of x to be the matching segment even if x already has a value.

!_|x| = (ALTER!-TENTATIVE |x|) will tentatively alter the value of |x| to be the matching segment but if a failure backs up then the old value of |x| will be restored.

Gerry Sussman and I have developed the following scheme for looking up the values of identifiers in interpreted code. On the

4.1 page 63a

MECHANISM OF IDENTIFIER LOOKUP



identifier stack when an identifier is bound the following information is stored:

1. the name of the identifier

2. the current value of the identifier

3. the place on the stack where the identifier was previously bound

Associated with each binding environment and identifier we have the place on the identifier stack where the identifier was last bound.

4.1.2 Syntax of Expressions

MATCHLESS uses Polish prefix notation for function calls with the actual call delimited by < and >. Of course we use the characters (and) to delimit lists. We use the characters [and] to delimit vectors. For example <+ 2 3> evaluates to 5. If y has the value 4, then <+ .y 1> will only match 5. The value of (.y) is (4) and the value of (<+ .y 1> (4 a) .y) is (5 (4 a) 4). If the function call is to denote a segment then it is delimited by { and }. The function REST will return the rest of the list that it is given as an argument. For example <rest (a b c)> evaluates to (b c). But (1 {rest (a b c)} e f) evaluates to (1 b c e f). Furthermore, (a b {rest (1 (e f) g)} k) will only match (a b (e f) g k). The components of lists, vectors, and nodes can be selected by subscripting. For example <2 (a b c)> evaluates to b and <3 [(a) e 5]> evaluates to 5. The expression <get |i| |x|> will return the location of the |i|th component of the structure |x|. Other values are computed from patterns. The value of [.y (a b) .y] is [4 (a b) 4]. Tuples are stored in the stack whereas the vectors are garbage collected. Lexically the scope of a tuple is the smallest enclosing pair of < and > or { and }. Otherwise vectors and tuples are indistinguishable. An argument of a function may be computed in parallel with the other arguments by delimiting the argument with |< and > instead of < and >. For example 7+3 could be computed in parallel with 2+4 in the expression <* |<+ 7 3> <+ 2 4>>. An argument of a function MUST be able to be computed in parallel if it is delimitted by !|< and >. In other words, if one branch becomes blocked the other must be able to continue execution.

4.2 Types

The type hierarchy is:

<?> for the universal type.

<WORD> for primitve types which are not pointers.

FALSE for the logical type false. All other data are considered to be true in conditional expressions. The null function call <> will evaluate to #FALSE.

CHARACTER for a character such as !"a or !"U. Again we are using ! as an escape character. The ! converts " into the quote for a single character.

<NUMBER> for numbers.

<FIXED> for fixed point number.
FIX for a small fixed point number.

BIG for a big fixed point number.

FLOAT for floating point number.

<POINTER> for pointers.

ATOM for atoms. The following are all atoms: a, foo, and hello

<STRUCTURE> for structured data. The operations of taking the REST of a structure and selecting the nth element are defined on all structures including tuples, vectors, lists, and nodes. For some structures the operations are more efficient because of special hardware.

TUPLE for a tuple of elements. Tuples are allocated from the stack of a process and are deleted on procedure exit. Tuples occupy contiguous blocks of memory. Once a tuple has been created its structure cannot be changed and its length can not be increased.

VECTOR for a vector. Vectors are allocated contiguous blocks of storage which are garbage collected when no longer pointed at. Although the structure of a vector cannot be changed, its length can be increased at the cost of a garbage collection. Otherwise vectors are identical to tuples.

STRING for a string. This is just a vector of characters. For example "ba", "3", and "a b" are strings

LIST for a list. Lists have the advantage over vectors that their structure can be changed after they have been created. They have the disadvantage that it takes a time proportional to n to get the nth element.

NODE for a node which has properties. Nodes are the most general form of structured data in the language. The others are included for reasons of efficiency for specialized structures. The components of a node are obtained by subscripting which is currently implemented by hash coding. A vector is approximately one third the size of its corresponding representation as a node.

The following types will not be explained here. They are included only for completeness. The complicated types and their abbreviations are:

JUNCTION for junction

ACTIVATION for activation.

STATE for state.

ARC for a node arc.

BIND for bindings.

<LOCATIVE> for a locative or generalized location.

VECTOR-LOCATIVE for a locative to an element of a vector.

TUPLE-LOCATIVE for a locative to an element of a tuple.

BINDING-LOCATIVE for a loactive to the value of an identifier

LIST-LOCATIVE for a locative to an element of a list.

LIST-REST-LOCATIVE for a locative to the rest of a list.

NODE-LOCATIVE for a locative to an element of a node.

LABEL for a label function.

PROCESS for process.

STACK for a stack

RING for a ring

ELEMENT-CALL for a element call.

SEGMENT-CALL for a segment call.

SEGMENT-VALUE-CALL for a segment value call.

4.3 Simple Examples of Matching

The idea of structural matching is fundamental to the MATCHLESS processor. By means of the primitive function <IS? |pattern| |expression|> we can determine if |pattern| matches |expression|. The function IS has the value true if the match succeeds and <> (which is FALSE) otherwise. Pattern matching takes place through the use of side effects to change the values of identifiers to be those of the objects which they match. The assignment statement in MATCHLESS is a variant of the primitive IS. The expression <_ |pattern| |expression|> is well defined only if |pattern| matches |expression|. The value of the function _ is the value of |expression|. Below we give some examples of matching where the values of identifiers are listed after assignment statements have been executed. We use the character - to delimit segments. For example the list (a b c) has subsegments:

--, -a-, -a b-, -a b c-, -b c-, -b-, and -c-.

The characters < and > are used to delimit function calls.

```
<prog [a [!=atom h] c]
        ;"This is a comment.
                We are inside a program in which we have
                declared a, declared h to be of type atom,
                and declared c"
        ;"in the test below
                the function IS will return true
                since the pattern (_a k _h !_c) matches
                the value ((1) k b o a)"
                <is? (_a k _h !_c) ((1) k b o a)>>
                a gets the value (1)
                h gets the value b
                c gets the value (o a)
```

The value of the program is true which is the value of the IS statement.

```
        <prog [c [!=atom h] a]
                ;"h is of type atom"
                <is? (!_c _h k _a) (a j b k q)>>
                c gets the value (a j)
                h gets the value b
                a gets the value q

        <prog [first last middle]
                <is? (_first !_middle _last) (a b c d)>>
                first gets the value a
                middle gets the value (b c)
                last gets the value d

        <prog [a b]
                <is? (_a _b) (d)>> fails because there is only one
element in (d).

         <prog [[!=atom a]]
                ;"a is of type atom"
                <is? _a (o t)>> fails because (o t) is not an atom.
```

An expression that consists of the prefix operator "." followed by a identifier will only match an object equal to the value of the identifier.

```
        <prog [a]
                <is? (!_a !.a) (a b c a b c)>>
                a gets the value (a b c)

        <prog [a b]
                <is? (!_a x !.a !_b) (a b x d x a b x d q)>>
                a gets the value (a b)
                a failure occurs because (!.a !_b) will not match (d x
a b x d q)
                a gets the value (a b x d)
                b gets the value (q)
```

An expression that consists of the prefix operator ? [the value given] followed by an identifier matches the value of the identifier if it has one, otherwise the identifier is assigned · value.

```
        <prog [a]
                <is? ?a t>>
                a gets the value t

        <prog [[!=fix [a 5]]]
                <is? ?a 4>>
                a is declared to be of type fix and initialized to 5
on entrance to the prog.  Consequently the assignment statement fails.

        <prog [a]
                <is? (!_a !?a) (a b c c b a)>> fails because once a is
assigned a value, a can only match a segment that is equal to the
value of a.
```

The function MATCH? is somewhat more powerful than the function IS? because it can match patterns against patterns.

```
        <prog [x y]
                <match ?x ?y>
                ;"link x and y by matching them to each other"
                <match ?x 3>
                ;"let x have the value 3 and thus set y to 3"
                .y
                ;"the value of y is the value of the prog"> evaluates
to 3
```

Restrictions on the value of an identifier can be acquired as the result of a match.

```
        <prog [x]
                <match ?x <less 5>>
                ;"x will only match numbers less than 5"
                <match 6 ?x>> fails since 6 is not less than 5
```

Side effects can propagate through structures:

```
        <prog [x y z]
                <match ?x [?y !?z]>
                <match (a b c) ?x>
                ;"y gets the value a and z gets the value (b c)">
```

4.4 Definitions of Procedures

4.4.1 Functional Procedures

<FUNCTION

+checker+ +activation-name+ [-function-declarations-] -expressions-> where +activation-name+ and +checker+ are optional, will evaluate to a function which will, when it is called, bind the formal parameters in the |function-declarations| to the actual parameters, evaluate the -expressions- returning the value of the last one as the value of the function. The +checker+ must be of the form <|procedure| -arguments-> for one value or {|procedure| -arguments-} for multiple values. The +checker+ is treated as a pattern that the values returned must match. The match is done so that any side effects are persistent. The |-function-declarations-| is of one of the following forms:

|arguments-specification| which may be one of the following:
[-formal-parameter-specifications-] where each formal-parameter-specification is of one of the following forms:

|evaluation-specification| where each |evaluation-specification| must be one of the following:
'|identifier| means that the |identifier| is to be bound to the write protected UNEVALUATED corresponding actual parameter.

|identifier| means that the |identifier| is to be bound to the VALUE of the corresponding actual parameter

[|attribute-specification| |evaluation-specification|] where the |attribute-specification| must be one of the following:

|attribute|

[-attributes-]

where each attribute must be one of the following:
¬"SPECIAL" means that the identifier may be used free in other modules. The symbol ¬"SPECIAL" is a unique string.

<|procedure| -arguments-> means that the identifier must always be either unassigned or bound to an object which matches the pattern <|procedure| -arguments->. The constraint is enforced by PLANNER. Any side effects of matching the pattern against the new value of an identifier are persistent.

[-formal-parameter-specifications- ¬"OPTIONAL" -optional-formal-parameter-specifications-]

where an |optional-formal-parameter-specification| is either a |formal-parameter-specification| or [|attribute-specification| [|evaluation-specification| |initial-value|]]. The ¬"OPTIONAL" construct is due to Chris Reeve. It alows for optional arguments and specifies how the identifier is to be initialized if the actual paramater is not present.

[-formal-parameter-specifications- ¬"REST" |identifier-specification|] which will bind the identifier in |identifier-specification| to the tuple of the rest of the arguments evaluated.

[-formal-parameter-specifications- ¬"REST" '|identifier-specification|] which will bind the identifier in |identifier-specification| to the write protected vector of the rest of the unevaluated arguments. The ' variant is due to Gary Peskin.

[¬"BIND" |identifier-specification| |arguments-specifcation| -declarations-] is used to first bind the identifier in |identifier-specification| to the bindings in effect when the function is invoked. In almost all cases use of ¬"BIND" can be avoided by reading the function into a local syntactic block so that no identifier conflicts can occur.

[¬"PATTERN" |calling-pattern| |arguments-specifcation|]
defines a calling pattern for pattern directed invocations.
The calling pattern is of the form [-declarations- |pattern|]
which declares identifiers for |pattern|.

For example:

```
        <<function [¬"rest" x] <2 .x>> 11 21 33> evaluates to 21
                since <2 [ 11 21 33]> is 21

        <<function [¬"rest" 'x] .x>
                a
                <+ 3 4>
                c> evaluates to  [a <+ 3 4> c]

        <<function     .x> 3> evaluates to 3

        <<function [x] .x> a> evaluates to a

        <<function !=fix [[!=fix x]] .x> <+ 2 2>> evaluates to 4 where
!=fix is <OF-TYPE fix>

        <<function !=fix [[!=fix x]] <+ .x 1>> 2> evalnates to 3

        <<function !=fix [[!=fix x] [!=fix y]] <+ .x .y>> 2 3>
evaluates to 5

        <<function [x ¬"optional" [y 3]] <+ .x .y>> 4> evaluates to 7

        <<function [x ¬"optional" [y 3]] <+ .x .y>> 4 5> evaluates to
9

        <<function [[!=fix 'x]] .x> 3> evaluates to 3

        <<function ['x] .x> a> evaluates to a

        <<function ['x] .x> <+ 2 2>> evaluates to <+ 2 2>
```

We would like to give a simple example of pattern directed invocation. Suppose that we have a sink s which we need unstopped. The classical solution is to know the name of a plumber which could be applied to the sink. Thus for example we might evaluate <plumber-

Perlman s>. The way we shall actually proceed is to advertise that we need a sink unstopped. Of couse we won't let just anyone work on our sink; he must come well reccmmended. For example he should be cheap and speedy. We will evaluate

```
<call
      [<[unstop s] $5>
       <speedy>]>
```

to offer to let some one unstop our sink for $5 provinding he is speedy. Now suppose that there are a few plumbers around:

```
<define plumber-Greenblatt
        <function
                  [¬"pattern"
                      [[sink] [unstop ?sink]]
                      fee]
               <cond
                  [<is <less $4> .fee>
                      <fail>]>
               ;"if the fee is less than $4
                      then fail"
               <Roto-Rooter .sink>
               ;"otherwise apply Roto-Rooter
                      to the sink">>

<define plumber-Perlman
        <function
                  [¬"pattern"
                      [[sink] [unstop ?sink]]
                      fee]
               <pour Drano .sink>
               ;"pour Drano in the sink"
               <send-bill <times 2 .fee>>
               ;"send a bill for twice the originally
                      agreed fee">>
```

To try to get our sink unstopped we might evaluate:

```
<prog [ ]
        <call
               [<[unstop s] $5>
```

```
                    <speedy>]>
                ;"advertise for a speedy plumber to
                      unstop sink s for $5"
                <cond
                   [<stopped-up? s>
                      <fail>]>
                ;"if the sink is still stopped up
                      then try again">
```

Suppose that both plumber-Greenblatt and plumber-Perlman are classified as speedy. Thus PLANNER will chose one or the other to invoke since both have patterns which match the calling pattern [unstop s]. If either one fails then the other will be tried. If one returns but the sink is still unstopped when he gets back then the mess the first created will be undone and the other tried.

We can define the function reverse which returns a newly constructed reverse of its argument as follows:

```
<define reverse <function [x]
        <rule [ ] .x
            |[<empty>
                .x]
            [<structure>
                <<storage .x> {reverse <rest .x>} <1 .x>>]
            ¬"else"
                <error>>>>
```

Thus <reverse [a [b c] 4]> is [4 [b c] a].

Functions with an arbitrary number of arguments are accommodated by passing a tuple which contains the evaluated arguments. Suppose that we already have a function PLUS which will add two numbers together.

```
<define + <function pl
        ;"let the name of the current activation be pl"
        [¬"rest" x]
        ;"we will receive a variable number of
                arguments in the tuple x"
        <for
                [[result 0] n]
                ;"initialize the identifier result to 0"
                        [[¬"test"
                                <is? [ ] .x>
                                <.pl .result>
                                ;"exit .pl with .result"
                        ;"each time before executing
                                the loop test to see
                                if x is a null tuple and if so then
                                return the result"]
                        [¬"step" <chop x>]
                        ;"after each pass through the loop chop x by
                                assigning x to the rest of x"]
                <_ :result <plus <1 .x> .result>>
                ;"the body of the loop is to add the first element of
                        x into the result">>>


        <+ 3 {rest (4 5 6)} 7> evaluates to 21

        <+ 3 2 4> evaluates to 9
```

<ACTOR-FUNCTION

[|object| |tail| |locative| |choice| -function-declarations-] -body-> is exactly like the function FUNCTION except for the folowing:

It is treated as an actor in pattern matching.

The first argument |object| is the matching object.

The second argument |tail| is a tail of the matching object or <> for an element call.

The third argument |locative| is a locative to |object| or <> if none such exists.

The fourth argument |choice| is not false only if the actor-function gets its choice how much to match.

The value of the actor-function is the rest of the object yet to be matched. Actor functions are useful as in internal interface between actors and functions.

4.4.2 Macro Procedures

Macros are expanded by the interpreter and by the compiler. The results are respectively interpreted and compiled. Macro procedures look like

<MACRO

|formal-parameters| -expressions-> The expansion of the macro is the value of the last expression. The character !' is used to suppress invocations. For example whereas <+ 2 2> evaluates to the NUMBER 4, !'<+ 2 2> evaluates to the function call <+ 2 2>.

```
<define choploc <macro ['x]
        !'<putloc
                .x
                !'<rest !'<in .x>>>>>
```

The macro choploc will take a location as its argument and cause the contents of that location to be changed to contain the rest of the previous contents.

<choploc <at y>> expands to <putloc <at y> <rest <in <at y>>>>

We could have defined the function + as a macro as follows:

```
 <define + <macro [¬"rest" 'x]
         ;"let x be the vector of unevaluated arguments"
         -rule .x
             [<empty>
                 ;"if x is <+> then the answer is 0"
                 0]
             #declare
             [[first rest]
                 ;"declare identifiers first and rest"
                 [:first !:rest]
                 ;"otherwise let first be the first argument and
                         rest be the rest of the arguments"
                 !'<plus .first !'<+ !.rest>>
                 ;"the answer is written out using
                         binary plus instead of +"]>>>
```

Thus

```
        <+ 3 2 4> expands to <plus 3 <plus 2 <plus 4 0>>>
```

4.4.3 Actor Procedures

Actors are used in patterns to match values. The primary difference between functions and actors is that functions produce values while actors match them. Actors and functions take their arguments in an exactly analogous fashion. Examples of actors are found in section 4.5 below.

<ACTOR

+checker+ +activation-name+ |function-declarations| -patterns->, where +activation-name+ and +checker+ are optional, evaluates to an actor which when it is invoked, matches an object which matches all of the -patterns- after the identifiers in the

|function-declarations| are bound. The |function-declarations| is interpreted EXACTLY as in FUNCTION.

```
        <<actor [¬"rest" x] <2 .x>> 1 a 3> matches only a

        <<actor [¬"rest" 'x] <2 .x>> a <+ 3 4> c> matches only <+ 3 4>

        <<actor [x] .x> 3> matches only 3

        <<actor [x] .x> a> matches only a

        <<actor !=fix [[!=fix x]] .x> <+ 2 2>> matches only 4 where
!=fix is <OF-TYPE fix>.

        <<actor !=fix [[!=fix x]] <+ .x 1>> 2> matches only 3

        <<actor !=fix [[!=fix x] [!=fix y]] <+ .x .y>> 2 3> matches
only 5

        <<actor [[!=fix 'x]] .x> 3> matches only 3

        <<actor ['x] .x> a> matches only a

        <<actor ['x] .x> <+ 2 2>> matches only <+ 2 2>
```

4.4.4 Type Procedures

Type procedures are used to define new types. New types can be defined by the union, direct product, and direct sum of already defined types. Types can also be defined as procedures by patterns.

```
        <define empty <either () [ ]>>
```

Define empty to be either an empty list () or an empty vector [ ].

```
<define monadic <either <number> !=atom <empty>>>
```

Define the type monadic to be a number or atomic or an empty structure.

```
<define property-list <actor <list> [ ]
                       <star (!=atom <?>)>>>
```

A property list is a list of two element lists whose first elements are atomic. The actor STAR is the Kleene star of regular expressions. For example the following are property lists: (), ((a (3))), and ((p1 4) (hello (r 3))).

4.4.4.1 Union of Types

<EITHER

-alternative-types-> is a type which must be one of the alternative types. For example we can define the type <number> to be the either <fixed> or the type of float. A disjunction of types expresses a constraint on what can be considered to be of the new class.

```
<define number <either <fixed> !=float>>

<prog [[<number> [x 3]]]
      ;"x is declared to be of type <number> and
            initialized to 3"
      <cond
          [<is? !=fix .x>
              yes]>>
evaluates to yes since x is really the of type fix
```

4.4.4.2 Product of Types

<PRODUCT

|type-name| |kind| |formal-parameters| -projection-specifications-> will create a type with name |type-name| made out of |kind| storage with |formal-parameters| as for functions and -projection-specifications-. Each |projection-specification| must be of the following form:

[|apparent-projector-names| [|initial| |pat|] +checker+ |actual-projector|] The |apparent-projectors-names| is either a single projector name or [|identifier| |list-of-projector-names|] where |identifier| ranges over |list-of-projector-names|. If +checker+ is present then only objects which match +checker+ can be stored in the component. When an instance is constructed, the elements are given the value |initial|. When an instance is decomposed, the pattern |pat| is used in matching. If only |initial| is given then |pat| is assumed to be the same as |initial|. If the actual projector is not specified then the next unused integer projector will be used. An actual projector which is a procedure call gives rise to a VIRTUAL projector storage for which is not necessarily physically present in the data structure. A product type can be RETRACTED to the |kind| of storage out of which it was constructed. The function PRODUCT grew out of some discussions that I had with Nick Pippinger.

```
        <define complex
                <product complex vector [r i]
                        [real [.r] <number>]
                        [imaginary [.i] <number>]>>
```

The type complex (for complex number) is the direct product of type <number> with projector real and type <number> with projector imaginary. The object complex is actually two procedures: a function which is the constructor and an actor which is the decomposer. Constructor-decomposers implement the overlap of functions and actors.

```
        <complex 3 4> evaluates to #complex [3 4] where # is the type
marker

        <retract <complex 3 4>> is [3 4].

        <getc real <complex 3 4>> (which computes the real component
of the complex number 3+4i) evaluates to 3

        <getc imaginary <complex 3 4>> evaluate to 4

        <prog [[<number> a b]]
                ;"This a comment. We are
                        inside a program. The identifiers a and b are
                        declared to be numbers"
                ;"in the assignment statement below
                        the pattern <complex _a _b> is matched
                        against the expression #complex [3 4]"
                <_ <complex _a _b> <complex 3 4>>>
        a gets the value 3
        b gets the value 4

        <getc real
                <_
                        <complex <replace 7> 4>
                        <complex 3 4>>>> evaluates to 7

        <prog [[!=complex [c <complex 1 2>]]]
                <getc real .c>> evaluates to 1
```

We need to be able to get at the locations of the components of a product. The <getc |projector| |structure|> is used for this purpose. The expression <PUTLOC |l| |x|> sets the location |l| to the value |x| and return the value |x|.

```
        <prog [[x <complex 3 4>]]
                ;"x is initialized to #complex [3 4]"
                <putloc
                        <getc real .x>
                        2>
                ;"x now has the value #complex [2 4]">
```

We can define a lower triangular matrix initialized with zeros as follows:

```
        <define triangular <product triangular vector [n]
                [
                        [i <thru 1 .n>]
                        [
                                <ivector .i <function [j] 0>>
                                ;"each component is initialized to
                                a zero vector of length i"]
                        <ivector .i>
                        ;"each component must be a vector
                                of length i"]>>

        <triangular 1> evaluates to #triangular [[0]]
        <triangular 2> evaluates to #triangular [[0][0 0]]
        <2 <triangular 2>> evaluate to [0 0]
```

We can define the type PDP-10 instruction as follows:

```
<define instruction <product instruction fix
                [op acc indir index addr]
                        [opcode [.op] !=fix <bits 9 27>]
                        [accumulator [.acc] !=fix <bits 4 23>]
                        [indirect [.indir] !=fix <bits 1 22>]
                        [index [.index] !=fix <bits 4 18>]
                        [address [.addr] !=fix <bits 18 0>]>>
```

A PDP-10 instruction has 9 bits of opcode which are 27 bits from the right end of the word, 4 bits for accumulator number, 1 bit to indicate indirection, 4 bits for index register number, and 18 bits for an address. An instruction with opcode 172 and 4 in the accumulator field causes the machine to halt. We can construct such an instruction with <instruction 172 4 0 0 0> which evaluates to #instruction 254400000000 in octal.

The next example illustrates the use of virtual components.

```
<define aobjn-ptr
        <product aobjn-ptr fix
            [l a]
                [length
                        [.l]
                        !=fix
                        <signed-bits 18 18>]
                [address
                        [.a]
                        !=fix
                        <bits 18 0>]>>
```

On a PDP-10 an aobjn pointer is is word whose left half contains the negative of the length of the rest of a vector and whose right half is the address of the element of the vector pointed at. The trailer is a virtual component which lies just after the vector. It can be defined as follows:

```
<define trailer <function [x]
        <get
                <+
                        <getc address .x>
                        <- <getc length .x>>
                        1>
                <getc address .x>>>>
```

<TYPE-VECTOR

-element-specifications-> constructs a type-vector where each element specification is of the form [|type| |value|] which initializes the apparent component |type| to |value|.

```
                <getc fix
                        <type-vector
                                [float "above"]
                                [fix "below"]>> evaluates to "below"
```

<CHARACTER-VECTOR

-element-specifications-> construct a character-vector where each element-specification is of the form [|character| |value|] which initializes the apparent component |character| to |value|.

```
                <putc
                        <character-vector
                                [!"a beginning]
                                [!"z end]>
                        [!"a very-beginning]>
                evaluates to
                #character-vector [[!"a very-beginning] [!"z end]]
```

4.4.4.3 Extension of Types

We need to be able to extend the types of values without otherwise altering them. For example 3 oranges are not the same as the fixed point number 3.

<EXTENSION

|type-name| |made-of|> will create a new type |type-name| which is an extension of |made-of|. We can define the type oranges by

```
        <define oranges <extension oranges  fix>>
Now <oranges 3> evaluates to #oranges 3.
```

<UNEXTEND

4.4 page 86a

DIRECT PRODUCT CONSTRUCTION

$$\begin{tikzcd}
 & \langle \text{COMPLEX} \rangle \arrow[dl, "\text{REAL}"'] \arrow[dr, "\text{IMAGINARY}"] & \\
\langle \text{NUM} \rangle & & \langle \text{NUM} \rangle
\end{tikzcd}$$

DIRECT SUM CONSTRUCTION

$$\begin{tikzcd}
 & \langle \text{FRUIT} \rangle & \\
\langle \text{FIX} \rangle \arrow[ur, "\text{APPLES}"] & & \langle \text{FIX} \rangle \arrow[ul, "\text{ORANGES}"']
\end{tikzcd}$$

|type-name|> returns the name of the type of which |type-name| is an extension. Thus <unextend oranges> evaluates to fix.

Individual elements of a given type can be retracted by the function RETRACT.

```
        <retract <oranges 3>> evaluates to the fixed point number 3
```

Similarly we can define apples by

```
                <define apples <extension appl fix>>
```

Then we can define fruit as the union of apples and oranges.

```
        <define fruit <either !=oranges !=apples>>

        <oranges 3> evaluates to #oranges 3 which is a <fruit>
```

<+ <oranges 3> <apples 4>> is an error because you can't add apples and oranges! To add apples and oranges the function + must be redefined in a local lexical block.

```
        <is? <fruit> <oranges 3>> is true

        <is? <fruit> 3> is <> (which is FALSE)
```

The actor <AS |pattern| |injector|> will be defined to match an object |obj| only if |obj| is of the type of the range of |injector| and |pattern| matches <RETRACT |obj|>.

```
        <prog [!=fix org]
                <is? <as :org oranges> <oranges 3>>>
                org gets the value 3

        <is? <as 4 apples> <oranges 4>> is <> (which is FALSE)
```

4.4.4.4 Direct Sums

The direct sum of types can be constructed as the disjunction of the extensions of the types.

4.4.4.5 Homogeneous Types

<HOMOGENEOUS

|n| |structure| |type|> will define |n| to be a homogeneous |structure| of |type|.

<homogeneous string vector character> defines the type string to be a homogeneous vector of characters.

<homogeneous big vector fix> define the type big to be a homogeneous vector of small fixed point numbers.

4.4.5 External Interrupts

The two kinds of external interrupts that are recognized are ATTENTION and ALARM interrupts. The current form of external interrupts is due to Peter Bishop. The attention handler is governed by the global value of the identifier HANDLER!-ATTENTION which must have the apparent components ¬"PROCESS" and ¬"HANDLER". If the ¬"PROCESS" component is <> then a running process will be interrupted. The initial attention handler is

```
!%<block (<oblist attention!-> <oblist>)>

<repeat out
        [¬"labels"
            [¬"special"
                [dismiss
                        <function [ ] <.out>>
```

```
                              ;"the label function dismiss will
                                   dismiss the interrupt"]]]
                    <print <eval <read>>>>

              !%<end-block>
```

The global value of the identifier ALARMS!-TIME is a write-protected list of alarm specifications. Each interrupt specification has the following apparent components:

¬"TIME" is the time after which the interrupt will occur. The interrupts specifications are stored in order of increasing time.

¬"IDENTIFICATION" is an object which identifies the alarm.

¬"HANDLER" is evaluated when the alarm goes off.

¬"PROCESS" is the process which is to be interrupted. If the component is <> then a running process is interrupted.

The global value of the identifier TIMERS!-RUNTIME is a write-protected list of timer specifications for all the PLANNER processes. The ¬"TIMERS" apparent component of each process is a similar write-protected list of timers for the for the runtime accumulated by that process. Each interrupt specification has the following apparent components:

¬"TIME" is the time after which the interrupt will occur. The interrupts specifications are stored in order of increasing time.

¬"IDENTIFICATION" is an object which identifies the alarm.

¬"HANDLER" is evaluated when the alarm goes off.

¬"PROCESS" is the process which is to be interrupted. If the component is <> then a running process is interrupted.

The above data structures can be modified using the following functions:

```
        <SET-ALARM
                |identification|
                |time|
                |handler|
                |process-to-be-interrupted|>
```

will set an alarm with |identification| which will go off after |time| interrupting |process-to-be-interrupted| with |handler|.

```
        <UNSET-ALARM
                |pattern-for-identification|
                |pattern-for-time|>
```

unsets all alarms whose identification matches |pattern-for-identification| and whose time matches |pattern-for-time|.

```
        <SET-TIMER
                |process|
                |identification|
                |runtime|
                |handler|
                |process-to-be-interrupted|>
```

sets a timer for |process| with |identification| which will go off after |runtime| interrupting |process-to-be-interrupted| with |handler|. If |process| is <> then the timer counts the time used for all processes.

```
        <UNSET-TIMER
                |process|
                |pattern-for-identification|
                |pattern-for-runtime|>
```

unsets all the timers for |process| whose identification matches |pattern-for-identification| and whose runtime matches |pattern-for-runtime|.

4.5 Functions in Expressions

4.5.1 [illegible]itions of Functions

Examples of the values of various expressions are given below:

a evaluates to a

(a b c) evaluates [illegible] a b c)

<+ 1 2> evaluates to 3

[3 {rest (a c)}] evaluates t[illegible] c]

(a b <+ 2 3>) evaluates to (a b 5)

(a b {quote (a b)}) evaluates to (a b a b)

If a has the value 3, then ([(.a)] b) evaluates to ([(3)] b)

4.5.1.1 Control Functions

4.5.1.1.1 Conditional

<UNFALSE

|x|> is the value of |x| if it is not false and fails otherwise.

```
<define unfalse
        <function [x]
                <cond
                   [.x]
                   [¬"else" <fail>]>>>
```

<OR?

-disjuncts-> evaluates each of the disjuncts in turn until one of them is not false in which case it is returned as the value of the function OR?. Otherwise the value of the function OR? is false.

```
!%<block (<oblist or!-> <oblist>)>

<define or? <function out [¬"rest" 'a]
        <repeat [[v <>]]
                <cond
                    [<empty? .a>
                        <.out .v>]>
                <_ :v <eval <1 .a>>>
                <cond
                    [.v
                        <.out .v>]>
                <chop a>>>>

!%<end-block>
```

<OR

-disjuncts-> is exactly like OR? except that if none of -disjuncts- is not false then a simple failure is generated.

```
!%<block (<oblist or!-> <oblist>)>

<define or
        <function [¬"rest" 'a]
                <unfalse <or? !.a>>>>

!%<end-block>
```

<AND?

-conjuncts-> evaluates each of the conjuncts in turn unless one of them is false in which case it returns the value false. Otherwise it returns the value of the last conjunct.

```
!%<block (<oblist and!-> <oblist>)>

<define and? <function out [¬"rest" 'a]
        <repeat [[v ¬"true"]]
                <cond
                   [<empty? .a>
                       <.out .v>]>
                <_ :v <eval <1 .a>>>
                <cond
                   [<not? .v>
                       <.out <>>]>
                <chop a>>>>

!%<end-block>
```

<AND

-conjuncts-> is exactly like AND? except that if one of the -conjuncts- is false then a simple failure is generated.

```
!%<block (<oblist and!-> <oblist>)>

<define and
        <function [¬"rest" 'a]
                <unfalse <and? !.a>>>>

!%<end-block>
```

<NOT?

|x|> is true if |x| is false and otherwise |x|.

```
<define not?
        <function [x]
                <cond
                  [.x <>]
                  [¬"else" ¬"true"]>>>
```

<NOT

|x|> is true if |x| is false and fails otherwise.

```
<define not
        <function [x]
                <unfalse <not? .x>>>>
```

<COND

+checker+ +activation-name+ -clauses-> is the conditional statement of the language. Each clause is of the form [predicate -body-] or of the form #DECLARE [[-declarations-] predicate -body-]. The predicate of each clause is evaluated in turn until one of them is not false. Then the rest of the elements of the clause are evaluated in turn with the value of the last element being the value of the function COND. If all the predicates are false then the value of the function COND is false. The function COND is due to John McCarthy.

<cond [<> 5]> evaluates to <>

<cond [<> 5] [¬"else" 6]> evaluates to 6

If the operator | is used in fron of a clause then the predicate of the clause may be evaluated before or after the predicate of the next clause or in parallel with it. The first predicate to converge to anything other than false wins the race. There are obvious timing errors in the indiscriminate use of | for clauses

<cond |[3 a] [4 b]> evaluates to either a or b

```
<CATCH
        +activation-name+
        [-declarations-]
        |x|
        [|k| ¬"TUPLE" |v|]
        -body->
```

establishes a catchpoint and then attempts to evaluate |x|. If the evaluation of |x| comes back with an abnormal exit then the catchpoint is removed, |k| is bound to the type of exit and -body- is evaluated. If control runs off the end of -body- then the abnormal exit is restarted. The abnormal returns which are currently defined are of the following argument tuples:

[¬"RESTORE" |activation| -values-] for a restoration of the failpoint |activation| with -values-.

[¬"EXIT-CALL" |f| |arguments|] for a non local exit call of |f|. The expressions |f| may be either an activation or a junction.

[¬"EXIT" |activation| |values|] for a non local exit to |activation| with -values-.

[¬"AGAIN" |activation|] for a non local reiteration of |activation|.

[¬"TERMINATE"] for a termination of the process

For example

```
        <prog [ ]
                <prog foo [ ]
                        <catch [ ]
                                <.foo 3 a>
                                ;"exit .foo with 3 and a"
                                [k ¬"rest" v]
                                <cond
                                    [<is? .k ¬"exit">
                                        <.bar
                                           <print
                                                (caught
                                                exiting
                                                with
                                                .v)>> ]>>>
                4>
```

prints (caught exiting with [3 a]) and then evaluates to 4

```
        <+
                <catch [ ]
                        4
                        [k ¬"rest" v]
                        <cond
                            [<is? .k ¬"fail">
                                <print "you can't get here!">]>>
                <print 5>
                <fail>>
```

prints 5 and then fails without printing anything more.

```
        <catch [ ]
                <+ <print 4> <fail>>
                [k ¬"rest" v]
                <cond
                    [<is? .k ¬"fail">
                        <print (caught failure)>]>>
```
will print 4, print (caught failure), and then continue failing.

```
        <FAILPOINT
                +checker+
                +activation-name+
                [-declarations-]
                |expr|
                [+messag+ +activation+]
                -body->
```
establishes a failpoint and then evaluates |expr|. If the evaluation does not produce a failure then the value of the function FAILPOINT is the value of |expr|. If the evaluation of |expr| or some subsequent evaluation ultimately fails back to the failpoint then the failpoint is disestablished, the identifier |message| is bound to the failure message, the identifier |activation| is bound to true if the failure is to a higher level activation, and -body- is evaluated.

```
                <failpcint [ ] <fail> [m a]
                        <print hello>>
                        prints hello and then restarts failing

                <failpcint [ ] 3 [m a]
                    <print 4>> evaluates to 3 but if a failure
                        ever backtracks to here
                        then 4 will be printed.

                <prog foo
                        <failpoint [ ] 9 [m a]
                                <.foo a>
                                ;"exit .foo with a">
                        <fail>> evaluates to a
```

```
        <RESTORE

                |activation| -values-> will restore the failpoint
```

named by |activation| and exit it with -values-. It is an error if |activation| is not the activation of a failpoint. The function RESTORE is due to Drew McDermott.

```
                <prog way-out [[a 3]]
                       <print
                              <failpoint out [ ] .a [a a]
                                     <cond
                                        [<is .a 5>
                                            <.way-out .a>]
                                        [-"else"
                                            <restore .out .a>]>>>
                       <inc!-persistent a>
                       <fail>> initializes a to 3, prints 3,
```
increments a to 4, fails back into the failpoint, restores the failpoint, prints 4, increments a to 5, fails back into the failpoint, and finally exits .way-out with the value 5. The following function does not represent good programming practice and is not original, but it does illustrate the use of RESTORE. The function <CHANCES |identifier| |exceeded|> will decrement the value of |identifier| each time a failure propagates through it until the value of |identifier| becomes less than or equal to zero at which point |exceeded| will be evaluated.

```
                !%<block (<oblist chances!-> <oblist>)>

                <define chances
                       <function ['i -"optional" ['e '<error>]]
                              <failpoint f [ ] <> [-"optional"]
                                     <_ :.i <- ..i 1>>
                                     <cond
                                        [<is <less= 0> ..i>
                                            <eval .e>]
                                        [-"else"
                                            <restore .f ..i>]>>>>

                !%<endblock>
```

<RULE

+checker+ +activation-name+ [-declarations-] |x| -clauses- --"ELSE"- -not-found-> where the +activation-name+ and +checker+ are optional gives a rule for the expression |x|. Each clause is of the form [|pattern| -body-] or of the form #DECLARE [[-declarations-] |pattern| -body-]. The value of |x| is matched against the pattern of each clause until a match is found. If there is only one element in the clause then the value of the function RULE is <> which is false. Otherwise the value is the value of the last element of the clause. If none of the patterns match then the value of the function RULE is the value of -not-found- if it exists or is <> (which is FALSE). If a clause is preceded by | then the |pattern| of the clause may be matched against |x| before or after the pattern of the next clause or in parallel with it. If more than one |pattern| matches then the first one to match wins the race.

```
                <rule [ ] 3 [!=fix]> evaluates to <> which is false
                <rule [x] a [_x (.x .x)]> evaluates to (a a)
                <rule [ ] c [d e]> evaluates to <>
                <rule [ ] h [1 a]  [h b] [3 c]> evaluates to b
                <rule [ ] <- 3 1> [1 b] [<+ 1 1> c] ¬"else" 5>
evaluates to c
                <rule [ ] a [b 3] ¬"else" 7> evaluates to 7
                <rule [ ] a [b 3]> evaluates to <> which is false
                <rule [ ] 5 |[<greater 3> "big"] [<less 7> "small"]>
could evaluate to either "big" or "small".
```

4.5.1.1.2 Block

<DECLARE

-declarations-> declares new top level local identifiers within the process which calls DECLARE. It returns a list of the identifiers declared.

```
        <ACTCB-CALLER

                |object|

                |tail|

                |locative|

                |choice|

                |pattern|

                |bindings-for-pattern|>
```

enables functions to call the pattern matcher to match |pattern| against objects efficiently in special cases.

```
        <CALL
                |junction-name|
                [<|f| -send-args-> |state-path-for-f| |recommendation-
for-f| ]
                |g|>
```

binds the identifier |junction-name| to the junction defined by CALL and then calls |f| with the specified arguments. The expression |f| may be any of the following:

a label function which will be invoked.

a process which will be resumed.

a function which will be invoked.

a port in which -send-args- will be queued.

an activation which will be exited.

a junction which will be invoked.

a pattern which will attempt a pattern directed invocation

The recommendation must be of one of the following forms:

[¬"USE" -pats-] says that function which matches one of the patterns -pats- MUST be used.

[¬"TRY" -pats-] sas that the functions which match the pattterns -pats- are be tried.

[¬"FILTER" |h|] syas that the functions

```
        <|h|
                <CANDICATES FUNCTION |f| |state-path-for-f|>>
```

are all to be invoked (possibley in parallel).

An ordinary function call <|f| -args-> is equivalent to

```
<CALL
        <|f| -args->
        <FUNCTION [y] .y>>
```

where y is arbitrary identifier. The form of the argument |g| as a function is due to Jerry Sussman. However, if |g| is of the form [|function| |state-path] then it allows for a pattern directed resumption through |state-path|. We can define a function idivide of n and d which returns the quotient and remainder of the integer division of n by d.

```
        <define idivide <function idivide [n d]
            <repeat
                        [[r .n] [q 0]]
                <cond
                    [<is? <less .n> .r>
                        <.idivide .q .r>
                        ;"exit .idivide with .q and .r"]>
                <_ :r <- .n .r>>
                <inc q>>>>
```

Now if we evaluate

```
        [a !{idivide 7 3!} b] evaluates to [a 2 1 b]

        <call
                <idivide 7 3>
                <function [a b]
                        <print .a>
                        <print .b>>> prints 2 and then prints
```

1

```
        <CALL
                |junction-name|
                <|f| -arguments->
                |g|
                |state-path|>
```

where |f| is a label procedure exits to the level where the label function |f| is defined and then invokes |f| with the specified -arguments-. The expression <|f| -arguments-> is an abbreviation for

```
                <CALL
                        <|f| -arguments->
                        <FUNCTION OUT [¬"TUPLE" X] <.OUT !.X>>>
```

Label procedures and junctions are generalizations of labels. Label procedures are defined using the ¬"LABELS" construct in block declarations. See the example under PROG. The function |g| is applied to the values received if the process which calls CALL is resumed. Executing <.|junction-name| -send-args-> will exit to the level where |junction-name| was defined and then invoke <|g| -send-args->. If the optional argument |g| is not present and |f| is defined in another process then the process which calls CALL is terminated.

```
        <TEMPORARY
```

|junction-name| <|f| -arguments> |g|> makes a CALL to |f| such that all the tentative side effects within the scope from the point of the call to the exit of |f| are undone.

```
                <prog [[x 0]]
                        <temporary
                                <<function [ ]
                                        <_ _x 4>
                                        ;"tentatively set x to 4">>>
                        ;"x is restored to 0 because the
```

```
                              call was temporary"
                      .r> evaluates to 0
```

<TEMPORIZE

|activation| -values-> exits |activation| with -values- undoing all the tentative side effects within the scope of |activation|.

```
        <prog [[x 0]]
                <prog out [ ]
                        <_ _x 4>
                        ;"tentatively set x to 4"
                        <temporize .out>
                        ;"exit the activation .out undoing all
                                the tenative actions in
                                the scope of the activation">
                .x> evaluates to 0

        <prog
                        [[x 0]
                        ¬"labels"
                        [f <function [ ] .x>]
                        ;"define f to be a label
                                function of no
                                arguments that returns the
                                value of x]
                <_ _x 4>
                ;"tentatively set x to 4"
                <temporize .f>
                ;"invoke the label function .f
                        undoing every thing which
                        is tentative that has been done since

                        was defined"> evaluates to 0

        <prog [[x 0]]
                <call out
                        <
                                <function [w]
                                        <_ _x .w>
                                        <temporize .out .x>>
                                4>
                        <function [y]
                                <print .x>
                                <print .y>>>> will print 0
                                and then print 4
```

<STRAIGHTEN

<|f| -arguments>> makes a CALL to |f| such that a simple failure will not be caught within the scope from the point of the call to the exit of |f|. The function STRAIGHTEN grew out of discussions that I had with Jeff Hill and Terry Winograd. A very similar concept is called "fast back" in parsing grammars. The expression !s|form| is an abbreviation for <STRAIGHTEN |form|>.

```
                <prog [[a 3] b]
                     <_
                          <vel _a _b>
                          4>
                     <print .a>
                     <cond
                        [<is? .a 4>
                          <fail>]>
                     .b> assigns a the value 4, prints 4, fails
back inside vel, restores a to have the value 3, assigns b the value
4, prints 3, and then finally evaluates to 4.
```

```
                <prog [[a 3] b]
                     !s<_
                          <vel _a _b>
                          4>
                     <print .a>
                     <cond
                        [<is? .a 4>
                          <fail>]>
                     .b> assigns a the value 4, prints 4, fails
back through _ since it has been straightened but restores a to have
the value 3 in the process, and thus the whole prog fails.
```

<STRAIGHTEN-UP

|activation|> straightens the investigation by setting up a failpoint which will convert a simple failure into <FAIL |activation|>.

```
                    <define straighten-up <function [activation]
                           <failpoint .activation
                               [message activa]
                                   <cond
                                       [<not? <or? .message .activa>>
                                           <fail <> .activation>]>>>>
```

<PERSISTENT

|junction-name| <|f| -arguments> |g|> makes a CALL to |f| such that all changees within the scope from the point of the call to the exit of |f| are persistent. That is they will not go away automatically by backtracking. The expression !p|form| is an abbreviation for <PERSISTENT |form|>.

```
                    <prog out [[a 3] b]
                           <failpoint [ ] <> [m a]
                                   <cond
                                       [<is? .a 3>
                                           <.out "win">]
                                       [¬"else"
                                           <print [a changed to .a]>
                                           <.out "lose">]>>
                           <_ <vel _a _b> 4>
                           <print .a>
                           <fail>> initializes a to 3, tentatively alters
a to 4, prints 4, fails back inside vel, restores a to 3, tentontavely
alters b to 4, print 3, fails back to the failpoint, notes that a is
still 3 and so exits the activation .out with "win".
```

```
                    <prog cut [[a 3] b]
                           <failpoint [ ] <> [m a]
                                   <cond
                                       [<is? .a 3>
                                           <.out "win">]
                                       [¬"else"
                                           <print [a changed to .a]>
                                           <.out "lose">]>>
                           !p<_ <vel _a _b> 4>
                           <print .a>
                           <fail>> initialized a to 3, alters a to 4,
prints 4, fails back into the failpoint, notes that a is no longer 4,
and so sexits the activation .out with "lose".
```

<IS?

|pattern| |expression| |is-table| |is-apply-table|> is true only if |pattern| matches the value of |expression|. The |is-table| must be a TYPE-VECTOR and so must the |is-apply-table|. The function IS matains two local identifiers TABLE!-IS and APPLY-TABLE!-IS which are respectively bound to |is-table| and |is-apply-table|.

<IS

|pattern| |expression| |is-table| |is-apply-table|> is true if |pattern| matches the value of |expression| and generates a simple failure otherwise.

<_

|pattern| |expression|> is an assignment statement. The value of the function _ is the value of |expression|.

```
<block (<oblist assign!-> <oblist>)>

<define
        <function ['pattern value]
                  <eval !'<is .pattern '.value>>
                  .value>>

<endblock>
```

<MATCH?

|pattern1| |pattern2|> is true if |pattern1| matches |pattern2| and is false otherwise.

<MATCH

|pattern1| |pattern2|> is true if |pattern1| matches |pattern2| and generates a simple failure otherwise.

<EVAL

|x| |b| |apply-table|> evaluates |x| using the bindings |b| to look up the values of identifiers and |apply-table| to apply objects according to their types. The |apply-table| must be a TYPE-VECTOR. The function EVAL maintains local identifiers TABLE!-EVAL and APPLY-TABLE!-EVAL to hold the current |eval-table| and |apply-table| respectively.

<QUOTE

|x|> is |x|. We may abbreviate <QUOTE |x|> as '|x|. For example <prog [[x 1]] '<+ .x 5>> evalutes to <+ .x 5>. Notice that according to the following definition QUOTE write protects its argument.

```
<define quote <function ['x] .x>>
```

<SUPPRESS

|x|> suppresses evaluation of the form |x|. We may abbreviate <SUPPRESS |x|> as !'|x|. For example <prog [[x 1]] !'<+ .x 5>> evaluates to <+ 1 5>.

<PROG

+checker+ +activation-name+ |declaration-specification| -body-> where the +activation-name+ and +checker+ are optional is a named program block. The |declaration-specification| is of the form

```
[-ordinary-declarations-
-"LABELS" -label-declarations
-"INTERNALS" -internal-declarations-]
```

where LABELS and INTERNALS are optional.

Each |internal-declaration| is of the following form:

[|f| <FUNCTION |formal-parameters| -body->] the identifier |f| is declared to be an internal function. As such it has special access to the local identifiers of the procedure to which it is internal. The identifier |f| may not have its value changed. The following constructs are very efficient within the procedure which delares |f| to be internal:

```
<|f| -arguments->
[|f| -arguments-]
!{|f| -arguments-!}
```

Internal functions provide a rapid means of common subexpression evaluation. The current form of internal functions is due to Peter Bishop and Dave Reed.

Each |ordinary-declaration| is of the following form:

[|attribute-specification| -bindings-] causes the identifers in the bindings to be rebound with the appropriate |attribute-specification|

where each binding must be one of the following two forms:

|identifier| indicating that the identifier is rebound and not assigned a value

[|identifier| |value|] which rebinds the |identifier| with an initial |value|

where |attribute-specification| must be one of the following:

|attribute| where each |attribute| must be one of the following:

|pattern| indicating that the value of the identifier must match |pattern|. A common pattern is <OF-TYPE |type-name|> (abbreviated !=|type-name|) which indicates that the value of the identifier must be of type |type-name|.

¬"SPECIAL" indicating that each of the identifiers is special meaning that it can be used as a free identifier in other procedures

[-attributes-]

Each |label-declaration| is of the form

[|f| |function|] so that execution <|f| -arguments-> will cause control to exit to the point where |f| was declared and |function| to be applied to the evaluated -arguments-.

If control falls through the bottom of the function PROG then it takes as its value the value of the last statement of the body. If called as a procedure a label exits to the activation in which the label was bound.

```
<prog foo [[x 1]
      ¬"labels"
            [nonfatal
                  <function [z]
                        <print (non-fatal .z)>
                        <_ :x a>
                        <again .foo>>]
            [fatal
                  <function [z]
                        <print (fatal .z)>
                        <.foo lose>
                        ;"exit .foo with
                              lose">]]
      ;"we have two label procedures
            nonfatal and fatal"
      <prog bar [[y .x] [x 1]]
            <cond
               [<is? .y 1>
                  <.nonfatal first-time>]
               [¬"else"
                  <.fatal second-time>]>>>
```

evaluates as follows:

```
foo is entered
x is initialized to 1
the labels fatal and nonfatal are bound
      bar is entered
```

```
                y is initialized to 1
                x is initialized to 1
                <.nonfatal first-time> is invoked
                        causing us to
                        exit BAR
        (non-fatal first-time) is printed
        x is changed to a
                bar is entered
                y is initialized to a
                x is initialized to 1
                <.fatal second-time> is invoked
                        causing us to exit FOO
        (fatal second-time) is printed
        foo is exited with the value lose
```

```
<BLOCKBIND
        +checker+ +activation-name+ [-declarations-]
        [¬"BIND" |block-bindings|
                |relative-bindings|
                |block-name|
                |block-declarations|
                |againer|]
        -body->
```

is exactly like PROG except that before the -body- is executed a name |block-name| and a set of block style (e.g. PROG, FOR, etc.) bindings |block-declarations| are established using |relative-bindings| to look up the values of any free identifiers. The resulting binding environment is bound to the identifier |block-bindings|. If <AGAIN |block-name|> is called, then |againer| is invoked. The function BLOCKBIND is useful for writing interpreters. We could define REPEAT as follows:

```
<define repeat
    <function p2
        [¬"bind" b1 ['name 'decs ¬"rest" "bd]]
        ;"let b1 be the bindings before p2 was entered
                and let name be the name of the repeat,
                decs be its declarations,
                and bd be its body"
```

```
                    <blockbind p1 [[iter .bd]]
                            ;"let iter be the rest of the body
                                 to be evaluated"
                              [-"bind" b2
                                   .b1
                                   .name
                                   .decs
                                   <prog [ ]
                                        <_ :iter .bd>
                                        ;"if <again .name>
                                             is executed,
                                             then reinitialize
                                             iter"
                                        <again .p1>>]
                            <cond
                              [<empty? .iter>
                              ;"if the body .iter is empty"
                                   <_ :iter .bd>
                                   ;"set iter to be the whole body"
                                   <again .p1>]>
                            <eval <1 .iter> .b2>
                            <chop iter>
                            ;"set the body iter to the rest of itself"
                            <again .p1>>>>
```

```
          <PROCBIND
                    +checker+ +activation-name+ [-declarations-]
                    [-"BIND" |procedure-bindings|
                             |relative-bindings|
                             |procedure-name|
                             |procedure-declarations| ]
                    -body-> is exactly like BLOCKBIND except that it takes
```
procedure style declarations (e.g. FUNCTION and ACTOR) instead of PROG style declarations.

4.5.1.1.3 Escape

```
          <CALL
                    |junction-name|
                    <|activation| -values->
                    |f|
                    |state-path|>
```
leaves the activation |activation| with the given values. The

expression <|activation| -values-> is an abbreviation for

```
                    <CALL
                            <|activation| -values->
                            <FUNCTION [X] .X>>
```

where X is an arbitrary identifier. The function |f| is applied to the values received if the process which calls CALL is resumed. If the optional argument |f| is not present and +activation-name+ is defined in another process, then the process which called CALL is terminated.

<AGAIN

|junction-name| |activation| |f|> reiterates the |activation|. If |activation| is an activation in another process, then the process which calls AGAIN will apply the function |f| to the values with which it is resumed. If the optional argument |f| is not present and |activation| is defined in another process, then the process which called AGAIN is terminated. It is illegal to execute <AGAIN |activation|> until all the declarations of |activation| have been processed.

```
                    <prog foo []
                            <print 1>
                            <again .foo>> prints 1 and then prints
1,prints 1, etc.

                    <prog bar [[a <again .bar>] [b 3]]
                            <print (you can't get here)>> causes an error
```

<FAIL> generates a simple failure in the match.

<FAIL |message|> causes a failure with a |message| to be reported above. A failure with a message can be caught only by the function FA: :CINT which is explained above.

<FAIL

|message| |place| |f|> generates a failure to |place| and then a failure with a |message| from there. The |place| may be either a process or an activation. The function |f| is applied to any arguments received by being resumed by another process. For example down inside a function whose activation is |a| and which has been called with a pattern directed invocation executing <fail ¬"caller" |a|> will signal that none of the other alternative functions should be tried.

4.5.1.1.4 Repetition

<REPEAT

+checker+ +activation-name+ [-declarations-] -body->
where the +activation-name+ and +checker+ are optional executes the body repeatedly until the body is exited by calling one of the functions CALL or AGAIN. Iterative programming in terms of repeats has the advantage that all loops are necessarily nested. The repeat loop may be exited with the value x by <.+activation-name+ x> where +activation-name+ is the name of the repeat loop. Executing <AGAIN .+activation-name+> after -declarations- have been processed transfers control to the first element of -body-.

```
<FOR
        +checker+ +activation-name+ [-declarations-]
        [[¬"INITIAL" -initial-action-]
        [¬"STEP" -step-action-]
        [¬"TEST" |predicate| -test-action-]]
        -body->
```

where the +activation-name+ and +checker+ are optional is defined to be an abbreviation for the following:

```
        <PROG +checker+ +activation-name+ [-declarations-]
                -intial-action-
                <REPEAT [ ]
                        <COND
                            [|predicate|
                                -test-action-
                                <.+activation-name+ <>>
                                ;"exit .+activation-name+ with <>"]>
                        -body-
                        -step-action->>
```

The FOR loop may be exited with the value |x| by <.+activation-name+ |x|> where +activation-name+ is the name of the FOR loop. Executing <AGAIN +activation-name+> jumps to the point labeled AGAIN in the expansion above. Alternatively, we have

```
        <FOR +checker+ +activation-name+ [-declarations-]
                [[¬"INITIAL" -initial-action-]
                [¬"TEST" |predicate| -test-action-]
                [¬"LIST" |item| ¬"IF" |condition|]
                [¬"STEP" -step-action-]]
                -body->
```

where the +activation-name+ and +checker+ are optional is like the FOR loop previously described except that the value of the for statement is the list of all the items such that the condition is true. It is equivalent to the following although it is implemented much more efficiently because it only does one cons for each item in the value.

```
<FOR
   +checker+
   +activation-name+
   [-declarations-
        [COLLECTED ()]]
        ;"declare COLLECTED to be initialized to ()"
        [[¬"INITIAL" -initial-action-]
        [¬"TEST"
                 |predicate|
                 -test-action-
                 <.+activation-name+ .COLLECTED>
                 ;"exit .+activation-name+
                         with .collected"]
        [¬"STEP"
                 <COND
                     [|condition|
                         ;"add |item| onto the end of
                                 COLLECTED if condition
                                 is met"
                         <_
                                 :COLLECTED
                                 (!.COLLECTED
                                 |item|)>]>
                 -step-action-]]
                 -body->
```

In addition to being able to list the elements produced we can append or concatenate them.

```
<FOR +checker+ +activation-name+ [-declarations-]
       [[¬"ON" |pattern| |value| -final-action-]]
-body->
```

where the +activation-name+ and +checker+ are optional executes the body of the loop once for each time that pattern matches value, <REST value>, <REST value 2>, etc. until <REST value n> becomes empty.

```
<FOR
   +checker+ +activation-name+
   [-declarations- [X |value|]]
          [[¬"TEST"
                 <IS? <EMPTY> .X>
                 ;"if X is empty then quit"
```

```
                                  -final-actions-]
                              [¬"STEP"
                                  ;"set X to the rest of X"
                                  <CHOP X>]]
                          <COND
                              [<IS? |pattern| .X>
                                  -body-
                                  ;"if |pattern| matches X
                                         execute -body-"]>>
```

```
        <FOR +checker+ +activation-name+ [-declarations-]
                [[¬"IN" |pattern| |value| -final-action-]]
        -body->
```

where the +activation-name+ and +checker+ are optional executes the body of the loop once for each time that pattern matches <1 |value|>, <1 <REST |value|>>, <1 <REST |value| 2>>, etc. until <REST |value| n> becomes empty. The ¬"IN" variant of a FOR loop was invented for LISP II. The above expression is equivalent to:

```
                <FOR +checker+ +activation-name+
                          [-declarations- [X |value|]]
                              [[¬"TEST"
                                          <IS? <EMPTY> .X>
                                          ;"if X is empty then quit"
                                          |final-actions|]
                              [¬"STEP"
                                          ;"set X to the rest of X"
                                          <CHOP X>]]
                          <COND
                              [<IS? |pattern| <1 .X>>
                                  -body-
                                  ;"if |pattern| matches the
                                          first element of X
                                          then exucute -body-"]>>
```

For example we can define a function which returns the reverse of a list as follows:

```
                <define reverse <function rev [x]
                          <for [first [answer ()]]
                                     [[¬"in" :first .x]
```

```
                        [¬"final"
                             <.rev .answer>
                             ;"exit .rev with .answer"]]
                   <_
                             :answer
                             (.first !.answer)>>>>
```

Thus <reverse (a b c)> is (c b a) The following function returns a list of the fixed point numbers in its argument:

```
<define numbers <function [x]
        <for [[!=fix first]]
             [[¬"in" :first .x]
             [¬"list" .x]]>>>
```

Thus <numbers (4 a (3 4) 5.0 6 [3])> is (4 6).

```
<FOR +checker+ +activation-name+ [-declarations-]
     [[¬"INC" |j| ¬"BY" |i| ¬"UNTIL" |predicate|]]
     -body-> is equivalent to

     <FOR +checker+ +activation-name+ [-declarations-]
          [[¬"TEST" |predicate|]
          [¬"STEP" <INC |j| |i|>]]
               -body->
```

```
<FOR +checker+ +activation-name+ [-declarations-]
     [[¬"INC" |j| ¬"BY" |i| ¬"THRU" |limit|]]
     -body-> is equivalent to

     <FOR +checker+ +activation-name+
          [-declarations-
               [S <ABS |i|>]
               [L |limit|]]
          ;"S is the absolute value of the step
               size which is frozen on entrance
               to the FOR loop"
          ;"the limit L is also frozen
               on entrance to the FOR loop"
          [[¬"INC" |j| ¬"BY" .S
```

```
                                ¬"UNTIL"
                                <IS? <GREATER .L> |j|>]]
                        -body->



        <FOR +checker+ +activation-name+ [-declarations-]
                [[¬"DEC" |j| ¬"BY" |i| ¬"UNTIL" |predicate|]]
                -body-> is equivalent to

                <FOR +checker+ +activation-name+  [-declarations-]
                        [[¬"TEST" |predicate|]
                        [¬"STEP" <DEC |j| |i|>]]
                                -body->



        <FOR +checker+ +activation-name+ [-declarations-]
                [[¬"DEC" |j| ¬"BY" |i| ¬"THRU" |limit|]]
                -body-> is equivalent to

                <FOR +checker+ +activation-name+
                        [-declarations-
                                [S <ABS |i|>]
                                [L |limit|]]
                        [[¬"DEC" |j| ¬"BY" .S ¬"UNTIL" <IS? <LESS .L>
|j|>]]
                        -body->

        <FOR +checker+ +activation-name+ [-declarations-]
                [[¬"THRU" |limit|]]
                -body-> is equivalent to:

                <FOR +checker+ +activation-name+ [-declarations- [I
1]]
                        [[¬"INC" I ¬"THRU" <ABS |limit|>]]
                        -body->
```

4.5.1.1.5 Multi-Process

Often it is convenient and more efficent to have more than one MATCHLESS process in existence at one time. By a process we mean a program counter together with a stack. Primitives are needed for the following functions:

1. Creating processes
2. Causing them to run
3. Terminating processes
4. Interrupting processes
5. Single stepping processes

<STEP

|p| |n| |condition|> executes the process |p| for |n| elementary steps unless the |condition| is met in which case it returns immediately. The value of the function STEP is the number of elementary steps actually executed in the process |p|. The existence of the function STEP means that PLANNER functions are not necessarily MONOTONE in the sense of lattice theory. A function f will be said to be CONTAINED in a function g if whenever <f x> converges then <g x> converges and furthermore <f x> = <g x>. A function h will be said to be MONOTONE if whenever x is contained in y then, <f x> is contained in <f y>.

<INVOKE

|junction-name| |p| |n| |condition| |f|> executes the process |p| through |n| complete procedure invocations unless the |condition| is met in which case the value is the number of

invocations completed. In this case |condition| is a function which is applied to the values returned by the invocation. After the invocations of |p| are complete control returns to the original process where |f| is applied to the values returned by the last invocation in |p|.

<PROCESS
|f| |top-activation| |scheduler|> creates a new process which begins execution with the |f|. The expression <PROCESS> returns the name of the process in which it is executed. Processes enable us to have multiple loci of control. We can hold our place in the problem solving process in some of the processes while advancing others. If |f| is a function then the process expects to be resumed with arguments for |f| the first time that it is entered. If |f| is of the form [|g| |port|] then it will hang on |port| and apply the the function |g| to the container of values that it extracts from |port|. The |top-activation| specifies how much of an existing process must be copied to start off the new process. Copying a process enalbles us to preserve its current state and still allow it to continue exectuon. The process is scheduled by the process |scheduler|. The value of the function PROCESS is the name of the created process. The garbage collector will terminate a process before it collects the storage for the process. If a process returns or fails off its top then it is terminated. The function |f| can handle normal returns and failures as it pleases. A process has the following apparant components:

¬"STATUS" is the status of the process. The status is be one of the following:

```
¬"RESUMEABLE" ¬"STOPPED" ¬"RUNABLE" ¬"RUNNING"
¬"TERMINATED"
```

¬"SCHEDULER" is the scheduler of the process

¬"RUNTIME" is the runtime charged to the process.

¬"TIMERS"is a list of timers for the process. The structure of a timer is explained above in the section on interrupts.

```
<CALL
        |junction-name|
        <|p| -send-args->
        |function|
        |state-path|>
```

resumes execution of the process |p| with the arguments -send-args- from the point that control last left it and suspend execution of the calling process. When the process which was suspended by the CALL statement is itself later resumed then the arguments received are passed as parameters to |function|. If the optional argument |function| is not present then the process which called CALL is terminated. The expression <|p| -send-args-> is an abbreviation for

```
<CALL
        <|p| -send-args->
        <FUNCTON OUT [¬"TUPLE" X] <.OUT !.X>>>.
```

For example <<process foo> 2 a> causes <foo 2 a> to be executed in a new process.

An example of the use of more than one process is in computing the fringe of an expression. The fringe of an expression is defined to be the expression with all interior parentheses removed. For example the fringe of (a (b) c) is (a b c) and the fringe of ((a (((b) c)))) is (a b c). We conjecture that the problem cannot be solved in pure LISP without the use of the primitives CONS, LABEL, or FUNCTION. We would like to write an efficient program to test whether two s-expressions have the same fringe. The problem is analogous to testing whether two derivation trees for a context free grammar have generated the same string. The function fringe? is not intrinsically interesting. Its importance lies in that fact that very similar control problems arise when a problem solver is trying to extract information from two different areas of investigation at once. We would like to be able to hold our place in one of the investigation spaces while we resume computation in the other. Multiple processes give us the capability which we need. The following symmetric form of the definition of fringe? is due to Bob Frankston.

```
<define fringe?
    <function cut [x y]
        <prog
            [[px
                <process tree-walk>
                ;"create a process which begins execution
                        with the function tree-walk"]
            [py <process tree-walk>]]
                <.px .x <process>>
                <.py .y <process>>
                <repeat [temporary]
                        <cond
                            [<==? <_ :temporary <.px>> <.py>>
                                <cond
                                    [<is? .temporary ()>
                                        <.out ¬"true">]>]
                            [¬"else" <.out <>>]>>>>>

<define tree-walk
        <function [x p]
                <.p>
                ;"the first thing to do is to resume
                        the main process with no arguments"
```

```
                                      <tree-walk1 .x .p>
                                      ;"after doing the complete
                                              tree walk resume the
                                              main process with the
                                              special value ()"
                                      <.p ()>>>

                <define tree-walk1
                        <function [x p]
                                <cond
                                    [<empty? .x>
                                        ;"if the structure is
                                                empty then return
                                                and try to find another atom"]
                                    [<is? !=atom .x>
                                        ;"resume the main process with the
                                                atom we have found"
                                        <.p .x>]
                                    [¬"else"
                                        <tree-walk1 <1 .x>>
                                        ;"find the atoms in the
                                                first element of .x"
                                        <tree-walk1 <rest .x>>
                                        ;"find the atoms in the rest
                                                of .x and then
                                                return to finding atoms on
                                                the remaining branches"]>>>
```

```
        <PORT>
```

creates a structure which contains two compontents:

> EXPORTS!-PORT is a ring which holds a queue of containers of exports waiting in the port.
>
> IMPORTERS!-PORT is a ring which holds a queue of processes waiting to take containers out of the port.

At any time either or both rings may be empty. Our concept of a port is derived from Rudy Krutar, Bob Balzer, and innumerable operating systems. The idea is that the port acts as a channel through which commerce may be transacted with some processes exporting through it and others importing what the others export. The commerce is completely containerized. An expression <CALL <|port| -values->> will place -values- in a container in |port|. When a process imports from a port it will get one container of values to apply to a function. Empty containers are allowed in which case the function of the importer will be passed no arguments.

Another example of the use of multiple processes occurs where there are two line printers and a number of processes which would like to get expressions printed. Suppose that <PORT-TO-PRINTERS> is the port to which things to be printed are exported. Furtermore let <PRINT-CHANNEL1> and <PRINT-CHANNEL2> be the channels for printer1 and printer2 restpectively.

```
                    <define printer
                        <function [print-channel]
                            <repeat [ ]
                                    ;"remove the next element
                                            from the print-port,
                                            print it
                                            on the print channel,
                                            and repeat"
                                    <call
                                        [ ]
                                        [
                                            <function [x]
                                                    <print
                                                            .x
                                                            .print-
channel>>
                                            <port-to-printers>]>>>>

                    <define setup-printers <function [ ]
                            <call
                                    <<process printer>
                                    ;"create a process for driving
                                            the first printer and pass it
                                            its print channel"
                                            <print-channel1>>
                                    [ ]>
                            <call
```

```
                                  <<process printer>
                                        <print-channel2>>
                                  [ ]>>>
```

After <setup-printers> has been called, then <<port-to-printers> |x|> will cause |x| to be queued and printed in its turn by one of the printers.

Now we would like to show how to do fringe? using ports instead of resumes.

```
        <define fringe?
            <function out [x y]
                <prog
                    [[port-x <port>]
                    [port-y <port>]
                    [px
                        <process [tree-walk .port-x]>
                        ;"create a process which begins execution
                              hanging on .port-x
                              with the function tree-walk"]
                   ;"at this point an activation of .px is waiting
                        in .port-x"
                    [py <process [tree-walk .port-y]>]
                   ;"at this point an activation of .py is waiting
                        in .port-y"]
                        <call
                              <.port-x .x .port-x>
                              ;"export .x .port-x to
                                    the port .port-x"
                              [.port-x]
                              ;"wait for a container of values
                                    from .port-x">
                        ;"at this point an activation of
                              .px is waiting
                              in .port-x"
                        <call <.port-y .y .port-y> [.port-y]>
                        ;"at this point an activation
                              of .py is waiting
                              in .port-y"
                        <repeat [temporary]
                              <cond
                                  [<==?
                                              |<_ :temporary
                                                 <call
                                                       <.port-x>
```

```
                                                    [.port-x]>>
                                             ;"the | allows the two
arguments of ==? to be computed in parallel and thus allows the
processes .px and .py to run in parallel to find the next atoms in .x
and .y"
                                             <call
                                                  <.port-y>
                                                  ;"export
                                                  an empty container
                                                            to .port-y"
                                                  [.port-y]
                                                  ;"wait for a container
                                                            on .port-y">>
                                        <cond
                                             [<is? .temporary ()>
                                                  <.cut ¬"true">]>]
                                   [¬"else" <.cut <>>]>>>>>

          <define tree-walk
                    <function [x p]
                              <call
                                        <.p>
                                        ;"export an empty container of values
                                                  to the port .p"
                                        [.p]
                                        ;"wait for a container of values
                                                  on the port p">
                              <tree-walk1 .x .p>
                              ;"after doing the complete tree walk
                                        export () on the port .p"
                              <call
                                        <.p ()>
                                        ;"insert () in the port .p"
                                        [.p]
                                        ;"wait for a container of values in
                                                  the port .p">>>

          <define tree-walk1
                    <function [x p]
                              <cond
                                   [<empty? .x>
                                        ;"if the structure is
                                                  empty then return
                                                  and try to find another atom"]
                                   [<is? !=atom .x>
                                        ;"resume the main process with the
                                                  atom we have found"
                                        <call
                                                  <.p .x>
                                                  ;"insert .x in the port .p"
                              [.p]
                              ;"wait for a container
                                     of values
                                     in the port .p">]
                    [¬"else"
                        <tree-walk1 <1 .x>>
                        ;"find the atoms in the first
                              element of .x"
                        <tree-walk1 <rest .x>>
                        ;"find the atoms in the rest of
                              .x and then
                              return to finding atoms on
                              the remaining branches"]>>
```

<WAIT-CALL

<|p| -send-args-> |function|> is exactly like CALL except that it is willing to wait until |p| becomes resumeable.

|<|p| -args-> might create a new process in which to evaluate <|p| -args-> in parallel with the normal order evaluation of the original process. The first | in the previous sentence is not metalinguistic. For example <* |<foo 3 4> <bar 3 5> |<+ .x 7> <g 2 2>> initiates evaluation of <foo 3 4> and possibly in parallel evaluates <bar 3 5>. After <bar 3 5> has been evaluated, it initiates evaluation of <+ .x 7> and possibly in parallel evaluates <g 2 2>. When all of the values have been computed, the function * is entered.

!|<|p| -args-> is exactly like |<|p| -args-> except that if one branch becomes blocked the other is guaranteed to be able to try to continue execution.

```
        <prog foo [ ]
            <+
                    !|<stop>
                    <.foo 3>
                    ;"exit .foo with 3">> evaluates to 3
```

<FORK

<|p| -args->> resumes execution of the suspended process |p| from the point that control last left it with the arguments -args- and in parallel continue execution of the calling process. It is an abbreviation for

<CALL <|p| -args-> [ ]>

For example <fork <<process foo> <bar> a>> causes <foo <bar> a> to be executed in a new process in parallel with the calling process. The value of the function FORK is |p|. The list of runnable processes is kept in the global value of the identifier RUNNABLE!-SCHEDULER. The initial scheduler is driven by the following handler for RUNTIME interrupts when a certain amount of runtime has elapsed:

```
!%<block (<oblist scheduler!-> <oblist>)>

<function [ ] <prog twiddle
    [victem [¬"global" runnable deserving]]
        ;"the processes that are still deserving to
              be run are kept in
              the identifier 'deserving'"
        <locker [ ] <getc lock schedule-queue>
              ;"lock the schedule variables while
                    they are being changed"
              <cond
                  [<empty? .deserving>
                      <_ :deserving .runnable>
                      <again .twiddle>]>
              <_ (:victem !:deserving) .deserving>
              <cond
                  [<is?
                      <getc ¬"status" .victem>
                      ¬"runnable">
                      ;"if the status is
                            runnable then
                            change it to running"
                      <putc
```

```
                                                        .victem
                                                        [¬"status"
¬"running"]>]
                                         [¬"else"
                                              <again .twiddle>]>
                               ;"this scheduler is strictly first in
                                       first out"
                               <continue .victem>
                               <locker [] <getc lock schedule-queue>
                                       <putc
                                              .victem
                                              [status ¬"runnable"]>>>>>
```

!%<end-block>

<TERMINATE

-processes-> causes -processes- to be stopped, their stacks unwound, their timers and alarms to be unset, and then put into a state such that they cannot later be resumed, interrupted, or continued. A process is automatically terminated when it returns or fails to its top level.

<STOP

|p|> stops the process |p| in such a way that it can later be continued or interrupted.

<CONTINUE

|p|> causes the process |p| to continue execution from where it was stopped.

<SUSPEND

|junction-name| |function|> suspends execution of the process which calls it. It is an abbreviation for

<CALL |junction-name| [] |function|>.

If the process is later resumed it begins execution by applying

|function| to the arguments received.

<INTERRUPT

|junction-name| |p| <|f| -arguments-> |g|> will interrupt the process |p| to evaluate the function |f| applied to -arguments- IN THE PROCESS |p|. If |f| returns normally then its values are given as arguments to |g|. Otherwise |g| will be applied to the arguments with which it is resumed. The primitive INTERRUPT allows the definition of functions which are not MONOTONE in the sense of lattice theory.

4.5.1.2 Data Functions

4.5.1.2.1 Specialists

4.5.1.2.1.1 Structure Functions

<STRUCTURE?

|x|> is true only if |x| is of storage type vector, list, stack, ring, or node.

```
<define structure?
        <function [x]
                <rule [ ] <storage .x>
                    [<either
                        vector
                        list
                        stack
                        ring
                        node>
```

```
                                        ¬"true"]
                                ¬"else"
                                   <>>>>
```

<EMPTY?

|x|> is true only if |x| is an empty structure.

```
<define empty?
        <function [x]
                <and?
                        <structure? .x>
                        <==? <length .x> 0>>>>
```

<MONAD?

|x|> is true only if |x| is not decomposable. In other words |x| is not a structure or it is empty.

```
<define monad?
        <function [x]
                <or?
                        <not? <structure? .x>>
                        <empty? .x>>>>
```

<CLOSURE

|procedure| |free-variables|> returns the closure of the |procedure| with the free variables bound to their values at the time when the closure is constructed. The CLOSURE primitive allows procedures to to have own variables. They enable us to easily construct generators such as those of GPS.

The function twice will take a function f as an argument and return a function which applies f to its argument twice.

```
<define twice <function [f]
        <closure <function [x] <.f <.f .x>>> f>>>
```

```
                <<prog [x 3]
                        <<closure <function [ ] .x> x>>>> evaluates to
3

                <prog [a [b 1]]
                        <_ :a <closure <function [ ] .b> b>>
                        <_ :b 2>
                        <.a>> evaluates to 1

                <prog [x 4]
                    <prog [
                                [y <closure <function [ ] .x> x>]
                                [x 0]]
                        <.y>>> evaluates to 4
```

Suppose that we wanted to define a generator |f| to be <elements |x|> such that each time that <|f|> is evaluated it returns a new element of |x|.

```
                <define elements <function [x]
                        <closure
                                <function [ ]
                                        <prog [[next <1 .x>]]
                                                <chop x>
                                                .next>>
                                x>>>
```

Now if we evaluate:

```
                <prog [[f <elements (a b c)>]]
                        <print <.f>>
                        ;"a is printed"
                        <print <.f>>
                        ;"b is printed">
```

<REST

|x| |n| +not-found+> returns the result of taking the rest of |x| |n| times. If the rest of |x| cannot be taken |n| times then +not-found+ is evaluated.

```
                <rest [a 4 d f] 2> evaluates to [d f]
                <1 <rest <node [1 a] [2 b]>>> is b
                <rest <rest [a 4 d f] 2> -1> is [4 d f]
```

If |n| is positive then, <rest <rest |x| |n|> <- |n|>> is an error or is identical to |x|. The function REST with a negative |n| may be applied only to tuple pointers, vector pointers, and node pointers. it may not be applied to list pointers.

<GET

|indicator| |object| +not-found+> returns the value under |indicator| for the |object| if such exists. Otherwise it returns the value of not-found. Integer indicators have special properties so that structures can be made out of lists, vectors, and nodes almost interchangeably. The expression <|integer| |object| +not-found+> is an abbreviation for <GET |integer| |object| +not-found+>.

```
<3 (a b c)> evaluates to c
<-1 <rest [a b c d] 3>> is b
<2 <rest <node [foo 1] [3 a] [2 b]>>> is a
<2 [a (b c) d]> evaluates to (b c).
<get foo
        <node [foo 1] [4 a]>> evaluates to 1
```

<GET!-NO-monitor

|indicator| |object| +not-found+> is exactly like get except that monitors for the location under |object| with arc name |indicator| will not be triggered.

<WAIT-GET

|indicator| |object|> is like GET except that if |object| does not yet have anything under |indicator| then the process

is suspended until |object| has something PUT under |indicator|.

<AT

|i| |o| +not-found+> returns the location of the value under the indicator |i| of the object |o|.

<putloc <at 2 [a 4]> 8> evaluates to [a 8]

<AT |o|> is the locative to the value of the identifer |o| if |o| is an atom and a locative to the rest of |o| if |o| is a list.

<ARC

|o| |indicator| +not-found+> is the arc from the object |o| with name |indicator| if there is one. Otherwise +not-found+ is evaluated.

<INITIAL

|o| +not-found+> is the initial node arc for the object |o| if it has one. Otherwise it returns the value of +not-found+.

<NEXT

|x| +nct-found+> returns the next arc after |x| for the object |o|, if there is one. Otherwise it returns the value of +not-found+.

<END? |o|> is true only if |o| is an end node with no leaves leaving it.

<LAST?

|x|> is true only if |x| is the last arc of the node.

<INDICATOR

|x|> is the indicator for the arc |x|.

<indicator <initial <node [a 3] [4 "r"]>>> is a

<HEAD

|x|> is the object at the head of the arc |x|.

<head <arc <put 3 [larger 2] [smaller 4]> smaller>> is 3

<TAIL

|x|> is the object at the tail of the arc |x|.

<tail <arc <node [a 3] [4 "r"]> a>> is 3

<LOCATIVE

|x|> is the location which holds the object at the end of the arc |x|.

<COPY

|x|> will completely copy |x|.

<==?

|x| |y|> is true only if |x| and |y| are identically the same object.

<=?

|x| |y|> is true only if |x| and |y| print the same as structures.

```
<define =? <function equal [x y]
        <equal1 .x .y .equal>>>

<define  equal1 <function equal1 [x y equal]
        <cond
            [<or? <monadic? .x> <monadic? .y>>
                 <cond
                    [<==? .x .y>
                        ¬"true"]
                    [¬"else"
                        <.equal <>>]>]
            [<==? <type .x> <type .y>>
                <repeat []
                        <cond
                            [<empty? .x>
                                <cond
                                    [<empty? .y>
                                        <.equal1 ¬"true">]
                                    [¬"else"
                                        <.equal <>>]>]
                            [<empty? .y>
                                <.equal <>>]>
                        <prog out []
                                <equal1
                                        <1
                                                .x
                                                <cond
                                                    [<has? 1 .y>
                                                        <.equal <>>]
                                                    [¬"else"
                                                        <.out>]>>
                                        <1
                                                .y
                                                <.equal <>>>
                                        <.equal>>
                        <chop x>
                        <chop y>>]
            [¬"else"
                <.equal <>>]>>>
```

<SIMILAR?

|x| |y|> is true only if |x| and |y| have similar values under their respective positive indicators. For example (3 "a4" [!"a]) is similar to [3 (!"a !"4) "a"].

```
<define similar? <function sim [x y]
        <similar1 .x .y .sim>>>

<define  similar1 <function sim1 [x y sim]
        <cond
            [<or? <monadic? .x> <monadic? .y>>
                <cond
                    [<=? .x .y>
                        ¬"true"]
                    [¬"else"
```

```
                        <.sim <>>]>]
          [¬"else"
               <repeat [ ]
                        <cond
                             [<empty? .x>
                                  <cond
                                       [<empty? .y>
                                            <.sim1 ¬"true">]
                                       [¬"else"
                                            <.sim <>>]>]
                             [<empty? .y>
                                  <.sim <>>]>
                        <prog out [ ]
                                  <similar1
                                            <1
                                                      .x
                                                      <cond
                                                           [<has? 1 .y>
                                                                <.sim <>>]
                                                           [¬"else"
                                                                <.out>]>>
                                            <1
                                                      .y
                                                      <.sim <>>>
                                            .sim>>
                        <chop x>
                        <chop y>]>>>
```

<ISOMORPHIC?

|x| |y|> is true only if |x| and |y| are isomorphic as graphs.

```
<define isomorphic? <function iso [x y]
                    <iso1 .x .y .iso>>>

<define iso1 <function [x y iso]
          <cond
               [<==? <type .x> <type .y>>
                    <prog out [ ]
                         <sub-iso1
                              <initial
                                   .x
                                   <.out>
                                   ;"if .x has no arcs then exit
                                        .out">
                              .y
                              .iso>>
                    <prog out []
                         <sub-iso1
                              <initial
                                   .y
                                   <.out>>
                              .x
                              .iso>>
                    ¬"true"]
               [¬"else"
                    <.iso1 <>>]>>>

<define sub-iso1 <function sub-iso-n [x-arc y iso]
          <repeat []
                    <iso1
                              <tail .x-arc>
                              <get <indicator .x-arc>
                                        .y
                                        <.iso <>>
                                        ;"f .y does not have a arc with
                                                  <indicator .x-arc> then,
                                                  exit .iso with <>">
                              .iso>
                    <_
                              .x-arc
                              <next
                                        .x-arc
                                        <.sub-iso-n>
                                        ;"exit .sub-iso-n if there are no
                                                  more x-arcs">>>>>
```

<PUT!-PERSISTENT

|object| -properties-> puts the properties on the |object|. A property of the form [|indicator| |v|] puts the value |v| under the |indicator|. A property of the form [|indicator|] deletes the |indicator| from the object. Integer indicators have special properties so that structures can be made out of lists, vectors, and nodes almost interchangeably.

<put <node [a 4] [3.5 c]> [a b] [3.5] [[e] 9]>

evaluates to #node [[a b] [[e] 9]]

<put (a 4) [1 "c"]> evaluates to ("c" 4)

Properties can be put on ANY of the data types of MATCHLESS. For example <put 3 [size small]> puts the value small under the indicator size for the fixed point number 3. The ability to associate any piece of data with any other piece is very useful. For example Gerry Sussman has pointed out that comments can be implemented in this way. The degree to which an expression has been simplified can be recorded. For example we might <put '<+ 3 4> [simplified canonically]> to indicate that '<+ 3 4> has been simplified canonically.

<PUT!-TENTATIVE

|object| -properties-> is exactly like PUT except that the properties of |object| are restored on backtracking.

<PUT!-NO-MONITOR

|object| -properties-> is exactly like PUT!-PERSITENT except that the monitors for the locations are not triggered.

<PUTREST!-PERSISTENT

|x| |y| |n| +not-found+> changes the REST of the list <rest |x| |n|> to be |y| where |y| must be a list. If <rest |x| <+ |n| 1>> is not a list then +nct-found+ is evaluated.

<putrest (3 a) (4 5)> evaluates to (3 4 5)

<PUTREST!-TENTATIVE

|x| |y| |n| +nct-found+> is exactly like PUTREST except that |x| is restored on backtracking.

```
<define putrest!-tentative <functicn
        [x ¬"optional"
               [Y ()]
```

```
                                    [n 0]
                                    [not-found !'<error>]]
                             <failpoint [[save <rest .x .n>]]
                                    <putrest .x .y .n .not-found>
                                    [¬"optional"]
                                    <putrest .x .save .n>>>>
```

<CHOP!-PERSISTENT

|x| |n| ↑not-found↑> assigns the identifier |x| the rest taken |n| times of its current value. The function CHOP was invented for a variant of LISP at MITRE.

```
!%<block (<oblist chop!-> <oblist>)>
<define chop
        <function
            ['x
                ¬"optional"
                [n 1]
                ['not-found '<error>]]
                   <_
                         :.x
                         <rest
                                  ..x
                                  .n
                                  .not-found>>>>
!%<end-block>

<prog [[v (1 2)]]
        <chop v>> evaluates to (2)
```

<CHOP!-TENTATIVE

|x| |n| ↑not-found↑> is like CHOP except that its results are not undone on backtracking.

```
!%<block (<oblist chop!-> <oblist>)>
<define chop
        <function
            ['x
                ¬"optional"
                [n 1]
                ['not-found '<error>]]
                   <_ _.x <rest ..x .n .not-found>>>>
!%<end-block>
```

<LENGTH

|x|> returns the length of the value of |x|.

```
<length (a b c)> evaluates to 3

<define length <function ln [x]
        <for [[n 0]]
            [[¬"in" <?> .x]
            [¬"final" <.ln .n>
                         ;"exit .ln with .n"]
                <inc n>]>>>
```

<INDEX

|x|> returns the rest index of |x|. The function INDEX is only defined for vectors and nodes.

<index <rest <rest [a e !"e e f g] 2> 3>> is 5

<TOP

|x| |n| +not-found+> is <REST |x| <- |n| <INDEX |x|>> +not-found+>

<BOTTOM

|x| |n| +not-found+> is <REST |x| <- <LENGTH |x|> |n|> +not-found+>

<UNIQUIZE

|value|> returns a pointer to the unique copy of |value|. The function UNIQUIZE can be used to save space and time in computations. The expression <UNIQUIZE |value|> may be abbreviated as !¬|value|. The function UNIQUIZE is due to Peter Bishop.

```
<uniquize "efg"> is ¬"efg"

<uniquize (a !"b ["e" 3]) > is  ¬(a !"b ¬[¬"e" 3])

<prog [[x [a b c]]]
       <==?
              <uniquize .x>
              <uniquize <copy .x>>>> is true.
```

<UNIQUELY?

|x|> is true only if |x| is a uniquely created copy of |x| i.e. to be <==? |x| <UNIQUIZE |x|>>.

<INCREASING?

-elements-> is true only if -elements- are arranged in increasing order in the the total ordering on unique expressions.

<SUBSTITUTE

|x| |pattern| |z|> substitute the value of |x| for all expressions in |z| that match |pattern|.

```
<substitute a !=atom (1 (x z))> evaluates to (a (a a))

!%<block (<oblist substitute!-> <oblist>)>

<define substitute <function
              [x 'p z]
       <subst
              .x
              <eval !'<actor [ ] .p>>
              .z>>>

<define subst <function [x p z]
       <rule [] .z
           [<.p>
              .x]
           [<monadic>
              .z]
           [<linear {?}>
              <<type .z>
                     <subst .x .p <1 .z>>
                      {subst .x .p <rest .z>}>]
```

```
                [<?>
                      .z]>>>

        !%<end-block>
```

<MEMBER?

|pat| |struc|> is the tail of |struc| whose first element matches |pat| if there is one and otherwise is <>.

<member? !=atom [3 4.5 (a) b 6 c]> evaluates to [b 6 c]

```
!%<block (<cblist member?> <cblist>)>

<define member?
        <function ['p s]
                <member1
                        <eval !'<actor [] .p>>
                        .s>>>

<define member1
        <function out [p s]
                <repeat []
                        <cond
                            [<is <empty> .s>
                                 <.out <>>]
                            [<is <.p> <1 .s>>
                                 <.out .s>]>
                        <chop s>>>>

!%<endblock>
```

4.5.1.2.1.1.1 List

<LIST!-CONSTRUCTOR

-values-> constructs a list of -values-. It is equivalent to (-values-).

4.5.1.2.1.1.2 Vector

Any expression enclosed within "(" and ")" evaluates to a list. Any expression enclosed within "[" and "]" evaluates to a vector.

<IVECTOR

|n| |fcn|> creates an implicit vector of length the value of |n| with entry i initialized to <|fcn| i>.

```
<define ivector <product vector  vector [n f]
        [[i <thru 1 .n>] [<.f .i>]]>>

<ivector
        3
        <function [i] .i>>
```

evaluates to [1 2 3].

<ITUPLE

|n| |fcn|> creates a definite tuple of length the value of |n| with entry i initialized to <|fcn| i>. A definite tuple can only be created as the initial value of an identifier in a declaration, as an element of a definite tuple, or as an argument to a function.

```
<INDEFINITE
        type [-declarations-]
        [-for-specifications-
        [¬"EXIT" |out-name|]
        [¬"ADJOIN" |expression|]]
        -body->
```
creates an indefinite tuple by setting up a for loop in which the elements of the tuple are generated element by element such that

condition is met. An indefinite tuple can only be created as the initial value of an identifier in a declaration, as an element of a definite tuple, or as an argument to a function. An indefininte tuple is a good way to pass arguments which are generated incrementally at run time. No tuples may be declared in -declarations-. Evaluating <.|out-name|> will cause INDEFINITE to return with the tuple generated.

```
                    <indefinite
                                        [[!=fix [i 1]]]
                                      ;"declare i to be a fixed point
                                              number initialized to 1"
                              [[¬"inc" i ¬"thru" .n]
                              ;"increment i thru .n"
                              [¬"adjoin" .i]
                              ;"each time through the loop adjoin the value
                                      of i to the tuple"]
                              ;"the body of the loop is empty">
```

evaluates to

[1 2 3 4] if the identifier n has the value 4

<UNSHARE

|x| |tail-of-x|> creates a copy of the value of |x| at the top level. The value of |tail-of-x| must be obtainable from the value of |x| by repeatedly applying the function REST. The value of the function UNSHARE is equal to its argument but it is not identical.

<unshare [1 x (y 2.0)]> evaluates to [1 x (y 2.0)]

<prog [[!=vector [x [a (4)]]]]

<is? <== <2 .x>> <2 <unshare .x>>>> evaluates

to true.

<VECTOR!-CONSTRUCTOR

-values-> constructs a vector of -values-. It is equivalent to [-values-].

4.5.1.2.1.1.3 String

```
        <STRING!-CONSTRUCTOR

                -values-> constructs a string of the -values-.

                <string
                        "Run"
                        " "
                        "Dick"
                        " "
                        "run"
                        !".>
```

evaluates to "Run Dick run."

4.5.1.2.1.1.4 Graph

```
        <NODE!-CONSTRUCTOR

                -properties-> constructs a node with -properties-.

        <SHARE
```

|node| |indicator| |locative|> will cause |node| to share the location under |indicator| with the location |locative|. The function SHARE is due to Peter Bishop.

4.5.1.2.1.1.5 Class

```
        <CLASS!-CONSTRUCTOR

                -elements-> will construct a class with -elements-.
```

4.5.1.2.1.2 Atom

<ATOM!-CONSTRUCTOR |string|> is the atom on the root oblist with print name |string|.

<ATOM!-CONSTRUCTOR

|string| |path| +not-found+> is the atom with the print name |string| in the |path| of oblists. If the optional argument +not-found+ is not present and there is not atom on |path| with print name |string| then a new atom is created in <1 |path|>.

<PNAME

|atom|> is the print name of |atom| which is a uninque string.

<pname hello!-dolly!-> is ¬"hello"

4.5.1.2.1.3 Word and Number Functions

<BITS

|s| |p|> defines a field of |s| bits that is |p| bits from the right end of the word.

<SIGNED-BITS

|s| |p|> defines a signed field of |s| bits that is |p| bits from the right end of the word.

<BYTE

|s| |p| |e|> returns a byte pointer to the byte of |s| bits that is |p| bits from the right end of the word pointed to by |e|.

<INC!-PERSISTENT

|var| |delta|> increments the value of the identifier |var| by |delta| and store the result in |var|. The body of INC will be in a separate lexical block so that identifier collisions cannot occur.

```
!%<block (<oblist inc!-> <oblist>)>
<define inc <function ['x]
        <_ :.x <+ ..x 1>>>>
!%<end-block>
```

<INC!-TENTATIVE

|var| |delta|> is like DEC except that |var| is restored in backtracting.

```
!%<block (<oblist inc!-> <oblist>)>
<define inc!-tentative <function ['x]
        <_ _.x <+ ..x 1>>>>
!%<end-block>
```

<DEC!-PERSISTENT

|var| |delta|> decrements the value of the identifier |var| by |delta| and store the result in |var|.

```
!%<block (<oblist dec!-> <oblist>)>
<define dec <function ['x]
        <_ :.x <- ..x 1>>>>
!%<end-block>
```

<DEC!-TENTATIVE

|var| |delta|> is like DEC except that |var| is restored in backtracting.

```
!%<block (<oblist dec!-> <oblist>)>
<define dec!-tentative <function ['x]
        <_ _.x <- ..x 1>>>>
```

```
!%<end-block>
```

<ASCENDING?

-elements-> is true only if -elements- are in ascending order. The function ASCENDING? is due to Gordon Benedict.

```
<define ascending? <function out [¬"rest" x]
        <cond
           [<is? <empty> .x>
                 <.out ¬"true">]
            [0"else"
                <repeat [ ]
                        <cond
                            [<is? <empty> <rest .x>>
                                 <.out ¬"true">]
                            [<not? <is?
                                    <greater <1 .x>>
                                    <2 .x>>>
                                 <.out <>>]>
                        <chop x>>]>>>
```

<DESCENDING?

-elements-> is true only if -elements- are in descending order.

<IDIVIDE

|dividend| -divisors-> computes the |quotient| and |remainder| of the |dividend| divided by the -divisors-.

```
[a !{idivide 7 3}} 69] evaluates to [a 2 1 69]

<call
        <idivide 11 4>
        <function [q r]
                <print .q>
                <print .r>>>
;"prints 2 and then prints 3"
```

<+

-numbers-> is the sum of -numbers-.

<+ 3 4 -5> is 2

<*

-numbers-> is the product of -numbers-.

<* 5 6> is 30

<ABS

|n|> is the absolute value of |n|.

<abs -3> is 3

<EXPT

|base| |exponent|> is exponentiation.

<expt 2 3> is 8

<-

|subtrahend| -subtractors-> is |subtrahnd| less -subtractors-.

<- 3 2> is 1

<- -5> is 5

<- 3 9> is -6

</

|dividend| -divisors-> is the floating point number |dividend| divided by -divisors-.

</ 4> is .5

</ 12 3> is 4.0

</ 3 2> is 1.5

</ 30 2 5> is 3.0

<MAX

-values-> is the maximum of -values-.

<max -3 <+ 4 .1> 4> is 4.1

<MIN

-values-> is the minimum of -values-.

4.5.1.2.1.4 Algebraic

<!+

-terms-> constructs the sum of the terms.

```
<!+
        '<* <expt x 2> 3>
        3
        '<* 2 x>
        '<* 4 x>
        4
        '<expt x 2>> evaluates to
<+
        7
        <* 6 x>
        <* 4 <expt x 2>>>
```

<!*

-factors-> constructs the product of the factors.

```
<!* 3 <!+ x 2> <!+ x -2> x> evaluates to
<+
        <* 3 <expt x 3>>
        <* -12 x>>
```

4.5.1.2.1.5 Locative

<IN

|location|> returns the contents of |location| as its value.

<prog [[x 1]] <in <at x>>> evalutates tc 1

<GENLOC

|x|> generates a new location (which is not on the stack) holding the location of |x|.

<in <genloc 3>> evaluates to 3

<PUTLOC!-PERSISTENT

|location| |value|> stores the |value| in the |location| and return the |value|. It is equivalent to <_ <smash |location|> |value|>.

<prog [x] <putloc <at x> 1>> assigns x the value 1

<PUTLOC!-TENTATIVE

|location| |value|> is exactly like PUTLOC except that |location| is restored on backtracking.

```
<define putloc!-tentative <function [location value]
          <failpoint
                    [[save <in .location>]]
                    <putloc .location .value>
                    [¬"optional"]
                    <putloc .location .save>>>>
```

<VALUE

|theta| |bindings|> is the value of the identifier which is the value of |theta|.

```
<prog [[¬"special" [x 1]] [y .x]]
          <value .y>> evaluates to 1
```

4.5.1.2.1.6 Stack

Stacks obey a last in first out storage discipline.

<STACK

+checker+> returns the name of a newly created stack to store elements of the appropriate +checker+.

<PUSH

|stack| -values-> pushs the -values- onto the |stack|. The value of PUSH is |stack|.

<POP

|stack| |number| +not-found+> pops |number| elements off |stack|. and returns them as the values of POP. The elements come off in the opposite order they went on.

```
(1 !(pop
        <push <stack> a b c d>
        e!)) evaluates to (1 d c b).
```

4.5.1.2.1.7 Ring

Elments can be inserted and removed from either end of a ring.

<RING

+checker+> returns the name of a newly created ring to store elements of the appropriate type.

<FRONT

|ring| |number| +not-found+> returns the front |number| elements of |ring|.

<REAR

|ring| |number| +not-found+> returns the rear |number| elments of |ring|.

<INSERT-FRONT

|ring| -values-> inserts -values- into the front of |ring|.

<INSERT-REAR

|ring| -values> inserts -values- in the rear of |ring|.

<DELETE-FRONT

|ring| |number| +not-found+> deletes |number| elements from the front of |ring| and returns them.

[a !{delete-front <insert-rear <ring> 1 2 3> 2!} b] is [a 1 2 b]

[a !{delete-rear <insert-rear <ring> 1 2 3> 2!} b] is [a 3 2 b].

<DELETE-REAR

|ring| |number| +not-found+> deletes |number| elements from the rear of |ring| and returns them.

4.5.1.2.1.8 Character

<CHARACTER>

matches any character.

<LOWER>

matches any of the twenty six lower case alphabetic characters.

<UPPER>

matches any of the twnety six upper case alphabetic characters.

<DIGIT>

matches any character which is a digit.

<ALPHABETIC>

matches any alphabetic character.

```
        <define alphabetic
                <actor [ ]
                        <either <lower> <upper>>>>
```

4.5.1.2.1.8 Input-output

Input-output is transacted through channels. The atomic names read in are looked up in directories called oblists.

<CHANNEL

|direction| |place| |place-dependent|> returns a communication channel in the |direction| specified to the |place| named. The |direction| may be either ¬"READ" or ¬"PRINT".

<CLOSE

-channels-> terminates transactions on the named -channels-.

<RESET

-channel-> resets |channel|.

<PRINC

|s| |channels|> prints |s| (which must be a string or charater) literally. It does not put quotes around it or otherwise translate |s|.

<PRIN1

|x| [-channels-] |path| |print-table| |macro-table|> prints the value of |x| on the output channels relative to |path| and returns it as the value of the function PRINT. The various types are printed according to the print functions which are defined in |print-table|. The function PRINT maintains three special identifiers: PATH!-PRINT, TABLE!-PRINT, MACROS!-PRINT, and CHANNELS!-PRINT. The |print-table| must be a TYPE-VECTOR. The |macro-table| must be a CHARACTER-VECTOR which has entries ¬"NEVER", ¬"BEGINING", or ¬"ALWAYS".

```
!%<block (<oblist print!-> <oblist>)>
<define prin1
        <function [x ¬"optional"
                [¬"special" [channels .channels]]
                [¬"special" [path .path]]
                [¬"special" [table .table]]
                [¬"special" [macros .macros]]]
                        <<getc <type .x> .table> .x>>>
!%<end-block>
```

The print function for vectors is:

```
<function out [y]
        <cond
            [<empty? .y>
                <princ "[ ]">
                <.out .y>]
```

```
                        [¬"else"
                            <princ !"[
                                    ;"print the open bracket which
                                            will be closed by ]">
                            <repeat [[x .y]]
                                    <prin1 <1 .x>>
                                    <chop x>
                                    <cond
                                        [<empty? .x>
                                            <princ
                                                    ;"close the ["
                                                    !"]>
                                            <.out .y>]>
                                    <princ !" >
                                    ;"print a space">]>>
```

<PRINT

|x| |path| |print-table| |channels|> prints a carriage return line feed, prints |x| and then prints a space. The |print-table| must be a TYPE-VECTOR.

```
<define print
        <function [x ¬"optional"
                [¬"special" [path .path]]
                [¬"special" [table .table]]
                [¬"special" [channels .channels]]]
                        <princ " ">
                        <prin1 .x>
                        <princ " ">>>
```

<OBLIST> is the root oblist.

<OBLIST

|trailer| +not-found+> is the oblist with the specified |trailer|. If the optional argument +not-found+ is not present and there is no oblist with |trailer| then one is created.

<TRAILER

|atom|> is the name of the oblist on which |atom|

exists. The trailer of an atom on the root oblist is <>.

<ON

|string| |path|> returns the first oblist in |path| on which an atom with print name |string| exists if there is one. Otherwise it returns <>.

<LINK

|atom| |path| |string|> creates link on the first element of |path| with print name |string|. It is an error if there is an atom with print name |string| already on |path|. Both |path| and |string| are optional.

```
                <prog [ ]
                        <link
                                top!-middle!-bottom
                                (<oblist me!->)
                                "tmb">
                        tmb!-me> evaluates to top!-middle!-bottom
```

<BLOCK

|path|> begins a new lexical block where atoms are looked up on |path|. The function BLOCK is due to Jerry Sussman.

<END-BLOCK>

closes the current lexical block restoring PATH!-READ to its previous value.

<READCH

|channel| +not-found+> removes the next character from |channel|. If there are none, then +not-found+ is evaluated.

<NEXTCH

|channel| +not-found+> is the next character in

|channel|. The channel is not modified by NEXTCH. If there are none then +not-found+ is evaluated.

<READ

|channel| |path| +not-found+ |macros| |syntax|>

returns the next expression from the input |channel| with atoms which are not on |path| created in the first element of |path|. The macro characters are as defined by functions of one argument in |macros| which must be of type VECTOR-OF-CHARACTERS. The argument of the function is the macro character which triggered it. The lexical syntactic class of each character is defined by |syntax| which also must be a CHARACTER-VECTOR. The idea for the read tables is due to John White If there are no more expressions on the channel, then +not-found+ is evaluated. The function READ maintains special local identifiers CHANNEL!-READ, PATH!-READ, NOT-FOUND!-READ, MACROS!-READ, and SYNTAX!-READ. from which it obtains the appropriate information. The definition of READ is:

```
        !%<block (<oblist read!-> <oblist>)>

        <define read
                <function [¬"optional"
                                [¬"special" [channel .channel]]
                                [¬"special" [path .path]]
                                [¬"special"
                                        ['not-found
                                                '<error ¬"end-of-file-
reached">]]
                                [¬"special" [table .table]]
                                [¬"special" [syntax .syntax]]]
                        <prog loop [character]
                                <_ :character <nextch>>
                                ;"let character be next charaacter
about
                                        to be read"
```

```
                                         <cond
                                            [<is? <getc .character .syntax>
¬"ignore">
                                                <again .loop>]>
                                         <<getc <readch> .table> .character>>
                                         ;"execute the read procedure
associated
                                                with the first character">>

       !%<end-block>
```

The following are the macro characters which are predefined for the reader:

#|type| |objec' reads |object| then tries to convert it to be a |type|.
For example #complex [3 4] will attempt to convert [3 4] to type complex.

!#FALSE is the unique object FALSE.

!#NODE +rest-index+ [-properties-] where each |property| is of the form [|indicator| |value|] is a node.

!#PROPERTIES |object| [-properties-] where each |property| is of the form [|indicator| |value|] is an object with properties.

!#ARC [|object| |indicator|] is a arc from |object| with name |indicator|.

!|character| is read as a single character.
The ! serves as an escape for characters which cannot be input directly.

!! is the exclamation character.
This is the only way to get in the character !.

%|form| reads |form| evaluate it and use the value as the expression read.
The % macro is due to Chris Reeve.

!%|form| reads |form| evaluate it and then pretend that what was actually read was the null string.
The !% macro is due to Chris Reeve.
The macro character !% enables us to have side effects while reading.
For example:

!%<block |path|> causes the reader to read the subsequent
items into |path| until the matching !%<end-block> is encountered.

$ terminates commands.

"|string|" is a character string.

!"|character| is a single character.

¢|character| reads |character| as though it were not a special character.
In other words ¢|character| is an ordinary alphabetic character to
the literal reader as though <getc |character| |syntax|> were ¬"alphabetic".

(~elements~) is a list.
The read function for !"(is:

```
<function [c]
    ;"the value will be a list"
    <list
        {indefinite [x]
                        ;"construct a tuple of indefinite size
                              made out of the values of x"
                        [[¬"adjoin" .x]
                        [¬"exit" out]]
            <call
                <read>
                <function [¬"rest" t]
                        <cond
                            [<is? <length .t> 2>
                                ;"read has returned with
                                      two values"
                                <rule [ ] <1 .t>
                                    [;"first of .t matches ("
                                    !")
                                        ;"the first value is
                                              a right paren"
                                        <.out>]
                                    ¬"else"
                                        <error
                                              "mismatched
left">>]
                            [¬"else"
                                <_ :x <1 .t>>]>>>}>>
```

The read function for !") is:

```
        <function out [c]
                ;"exit with two values so that any
                        function which
                        calls this one will know
                        something is fishy"
                <.out
                        ;"this should match ("
                        !")
                        <>>>
```

[-elements-] is a vector.

![-elements-!] is a homogeneous vector.
The notation is due to Chris Reeve and Gerry Sussman.

<-elements-> is an element form.

{-elements-} is a segment form.

!{-elements-!} is a multiple value segment form.

||form| is #ALLOW-PARALLEL |form|.

!||form| is #ESSENTIAL-PARALLEL |form|.

|atom|!- forces |atom| to be read into the ROOT oblist.

|atom|!-|trailer| reads |atom| into the oblist with |trailer|.
If the following is typed in:

```
        <prog [ ]
                foo!-thesis!-
                bar!-preface!-thesis!-
                !%<block (<oblist preface!-thesis!->
                        <oblist thesis!->)>
                (mumble hello!- foo bar 3 thesis preface)
                !%<end-block>>

    then it will evaluate to
                (mumble!-preface!-thesis
                hello
                foo!-thesis
                bar!-preface!-thesis
                3
                thesis
                preface!-thesis)
```

{-expressions- |element| ; |comment| -more-expressions-}
The read function for !"; is:

```
        <function out [character] <.out !"; <read>>>
```

{-expressions- |element| !; |intent| -more-expressions-}
The read function for !"!; is:

```
        <function out [character] <.out !"!; <read>>>
```

The following prefix macro characters are predefined.

'|expression| is <QUOTE |expression|>.
The ' macro is due to John White.
The read function for the character ' is:

```
        <function [character] !'<quote <read>>>
```

!'|form| is <SUPPRESS |form|> which suppresses invocation of |form|.
The read function for !"!' is :

```
        <function [character] !'<suppress <read>>>
```

¬|value| is a unique copy of |value|.
The read function for !"¬ is:

```
        <function [character] <uniquize <read>>>
```

!¬|value| is <UNIQUIZE |value|>.
The read function for !"!¬ is:

```
        <function [character] !'<uniquize <read>>>
```

!=|atom| is <OF-TYPE |atom|>.

&|form| is <GATE |form|>

!t<-elements-> is <TEMPORARY <-elements->>

!t{-elements-} is {TEMPORARY <-elements->}

!s<-elements-> is <STRAIGHTEN <-elements->>

!s{-elements-} is {STRAIGHTEN <-elements->}

!p<-elements-> is <PERSISTENT <-elements->>

```
!p{-elements-} is {PERSISTENT <-elements->}
The macro characters !t !s, and !p are due
to Peter Bishop.

.|identifier| is <VALUE |identifier|>.

!.|identifier| is {VALUE |identifier|}.

,|identifier| is <GLOBAL |identifier>.

!,|identifier| is {GLOBAL |identifier}.

_|identifier| is <ALTER!-TENTATIVE |identifier|>.

!_|identifier| is {ALTER!-TENTATIVE |identifier|}.

:|identifier| is <ALTER!-PERSISTENT |identifier|>.

!:|identifier| is {ALTER!-PERSISTENT |identifier|}.

?|identifier| is <GIVEN |identifier|>.

!?|identifier| is {GIVEN |identifier|}.
```

4.5.1.2.2 Protection

<UNPROTECT

|x| |u|> allows access to the object x according to the use |u| which may be:

```
¬"write" for write
¬"execute" for execute
```

Restricting the access of a piece of data ensures that it can not be used for a purpose which was not intended. For example it can be used to insure that checking routines do not modify the data which they are supposed to inspect for errors.

<PROTECT

|x| |u|> restricts the uses to which |x| may be put by not allowing the use |u| which may be ¬"READ", ¬"PUT", or ¬"WRITE. The use ¬"PUT" protects against putting on non-numerical indicators whereas ¬"WRITE" protects the numerical indicators.

```
<put
        <2 <protect (a  (3 4)) ¬"write">>
        [1 a]> causes a write protection error
```

<rest <protect (a 2 b) ¬"write">> returns (2 b) with write protect

<PROTECTION

|x|> returns a vector of the protection modes of access for |x|.

<protection <rest <protect (a 2 b) ¬"write">>> is [¬"write]

4.5.1.2.3 Monitoring

<MONITOR

|l| |f| |u|> monitors the location |l| with the function |f| for the use |u|. The use may be a list of any of the following:

```
¬"READ" for read
¬"EXECUTE" for execute
¬"WRITE" for write
```

If a process attempts to used a monitored location then <f |l| |u| |x|> is evaluated. If a write operation is being attempted, then x is the value which is being stored. If a execute operation is being completed for a function, then x is the tuple of values being returned. If an execute operation is being completed for an actor, then x is the object that was matched. Monitoring is implemented in a way that is logically equivalent to creating a arc from the location |l| to the list of monitors for the location |l| under the indicator MONITORS. Dave Reed invented the more efficient method that is actually used. Monitors are useful for implementing various kinds of procedural data. For example they are used to implement break points in the language. The following procedure will make a list (called history-of-x) of all the values that are stored into the special identifier x.

```
<monitor
        <at x>
        <function [l u v]
                <_ :history-of-x (.v !.history-of-x)>>
                ¬"write">
```

Next we would like to describe how monitors can be used to implement an idea due to Peter Landin which he calls a stream. The idea is that the elements of a list should be able to be dynamically computed instead of all of them having to be computed at once. For example in debugging the elements of a list might be computed incrementally as they are needed by being input from a teletypewriter. We could construct such a list |l| as follows:

```
<monitor
        (0)
        ;"the 0 is a dummy which will
                be replaced with the first
                element read"
        ,f
```

```
                    ¬"read">
```

Were we define f by:

```
                <define f
                   <function [l u v]
                      <monitor
                            <rest
                                  <_
                                        (<replace <read>>
                                              {replace (0)})
                                        .l>>
                            ,f
                            ;"monitor the rest of the list with f"
                            ¬"read">>>
```

Now <1 |l|> is the first expression read, <2 |l|> is the second, etc.

<UNMONITOR

|l| |pat|> unmonitors the location |l| by all functions that match |pat|.

4.5.1.2.4 Type

<RETRACT

|x|> returns the value |x| retracted to the type in which it was defined. The function RETRACT is the identity function on objects of primitive type.

<STORAGE

|x|> returns the primitive storage allocation type of |x|. The primitive storage types are LIST, VECTOR, STRING, HOMOGENEOUS-VECTOR, STACK, RING, ATOM, ACTIVATION, JUNCTION, LABEL, PROCESS, and NODE.

<TYPE

|x|> returns the dynamic type of |x|.

<DECLARED

|x|> returns the declared attributes of |x|. The function DECLARED is useful in deciding how to expand macros.

<GETC

|apparent-indicator| |object| +not-found+> gets the |apparent-indicator| component of |object| according to the structure definition for <TY |object|>.

<ATC

|apparent-indicator| |object| +not-found+> returns a locative to the |apparent-indicator| component of |object| according to the structure definition for <TYPE |object|>.

<PUTC!-PERSISTENT

|object| -properties-> puts -properties- on |object| according to the structure definition for <TYPE |object|>.

<PUTC!-TENTATIVE

|object| -properties-> is exactly like PUTC except that the properties of |object| are restored on bac racking.

4.5.1.2.5 Synchronization

<LOCK

-lock-specifications-> attempts to satisfy the -lock-specifications- where each lock-specification must be one of the following:

|location| means that |location| is to be locked if it is not already locked.

[¬"RELOCK" |location|] means that |location| is to be relocked even if it is already locked.

[¬"UNLOCKED" |location|] means that |location| must be unlocked.

The process which calls the function LOCK is suspended until all the -lock-specifications- are satisfied. Suppose that we have a data base that sometimes is momentarily in an inconsistent state while it is being modified. We would like to set up locks so that arbitrarily many processes can be reading the data base at one time but only one process can modify it at a time. Suppose that each data base has a READLOCK and a WRITELOCK component in addtion to a CONTENT component.

```
<define read-data-base <function rdb [data-base]
    <prog [current-content]
        <lock
                [¬"unlocked"
                        <atc writelock .data-base>]
                [¬"relock"
                        <atc readlock .data-base>]>
        ;"in order to read the data base
                the writelock must
                be off and the readlock
                must be relocked"
        <_ ;current-content <getc content .data-base>>
        <unlock <atc readlock .data-base>>
        ;"is done after the process stops reading"
        <.rdb .current-content>
        ;"exit .rdb with .current-content">>>
```

```
<define write-data-base
   <function [data-base new-content]
        <lock <atc writelock .data-base>>
        <lock <atc readlock .data-base>>
        ;"in order to write the data base the
writelock
                must be locked and
                then readlock must be locked"
        <putc .data-base [content .new-content]>
        <unlock
                <atc writelock .data-base>
                <atc readlock .data-base>>
        ;"is done after the process stops writing">>
```

<LOCKER

+checker+ +activation-name+ [-lock-specifications-] -body-> where the +activation-name+ a +checker+ are optional attempts to achieve -lock-specifations- execute the -body- and then unlock any locations that were locked by -lock-specifications-. The function LOCKER makes use of CATCH to insure that the locks are unlocked when +activation-name+ is exited. We can do the above example as follows:

```
<define read-data-base <function [data-base]
    <locker []
                [[¬"unlocked"
                        <atc writelock .data-base>]
                [¬"relock"
                        <atc readlock .data-base>]]
                <getc content .data-base>>>>

<define write-data-base
    <function [data-base new-content]
    <locker []
                [<atc writelock .data-base>]
        <locker []
                [<atc readlock .data-base>]
            <putc
                .data-base
                [content .new-content]>>>>>
```

<LOCKED?

-locations-> attempts to lock the locations which are arguments. If the locations cannot be locked then the function LOCKED? returns <>.

<UNLOCK

-locations-> unlocks the locations.

4.5.1.3 Debugging

<ERROR

|message|> will type out the message and go into an error loop.

```
!%<block (<oblist error!-> <oblist>)>

<define error <function
    [¬"optional" [message ¬"none"]]
    <print (¬"error-message:" .message) ,console>
    ;"print the message on the console channel"
    <repeat [¬"special" loop]
        [[old-out .out]
        ;"save the old value of out in old-out"
        [¬"special" [culprit <frame 3>]]
        ;"the culprit activation is the one three frames back"
        ¬"labels"
        [¬"special" [out <function [¬"optional" [n 1]]
                        ;"the label procedure out handles
                            exits from error loops"
                        <cond
                            [<is? <less 1> .n>
                                <again .loop>]
                            [¬"else"
                                <.old-out <- .n 1>>]>>]]]
        <print <eval <read>>>>>>

!%<end-block>
```

<DEBUG

|status|> will set the state of the debug state to |status|. The status may be ¬"on" or ¬"off".

<BINDINGS

|p|> is the current set of bindings for the process |p|.

<FRAME> is the current activation frame of the process which calls it.

<FRAME |place|> is the last activation of |place| if |place| is a process and is |place| if |place| is an activation.

<FRAME

|place| |n|> is the activation frame which is |n| frames back from |place|.

<PROCEDURE

|frame|> is the procedure of |frame|.

<NAME

|procedure|> is the name of |procedure| if it has one and <> otherwise.

<PROCNAME

|frame|> is the name of the procedure for |frame|. It is equivalent to <NAME <PROCEDURE |frame|>>.

<ARGS

|frame|> is the tuple of arguments of |frame|.

4.5.1.4 Identifier

<ASSIGNED?

|var| |b|> is true only if the identifier |var| has

been assigned a value within the bindings |b|.

<UNASSIGN

|var| |b|> makes |var| unassigned within the bindings |b|.

<BOUND?

|var| |b|> is true only if the identifier |var| is bound within the bindings |b|.

4.5.2 Examples of the Use of Functions

The function factorial is defined below in order to illustrate the syntax of functions that produce values. On entrance to REPEAT, temp is immediately bound to 1.

```
<define factorial
        <function  factorial [n] <repeat [[temp 1]]
                <cond
                    [<is? <less 1> .n>
                         <.factorial .temp>
                         ;"exit .factorial with .temp"]>
                <_ :temp <* .n .temp>>
                <dec n>>>>
```

Using a for statement, we can define factorial as follows:

```
<define factorial
        <function fact [n]
                <for
                        [[temp 1]]
                                [[¬"dec" n ¬"thru" 1]
                                [¬"final"
                                        <.fact .temp>
                                        ;exit .fact with .temp"]]
                        <_ :temp <* .n .temp>>>>>
```

Thus the value cf <factorial 3> is 6; and the value of <factorial <+ 2 2>> is 24

4.6 Actors in Patterns

Examples of actors are VEL for disjunction, NCN for negation, ALL for conjunction, and STAR for Kleene star in general regular expressions. We use the characters { and } to delimit actor calls that are to match as segments.

```
<prog [a b c]
        ;"we are inside a program.  we have declared the
                identifiers a b and c.
                In the assignment statement below the pattern
                (k {all _a _b} _c) is matched against
                (k x y z).
                The pattern {all _a _b} matches an expression
                only if both _a and _b match the expression."
        <is? (k {all _a _b} _c) (k x y z)>>
        a gets the value (x y)
        b gets the value (x y)
        c gets the value z

<prog [x c]
        <is? (!_x {either (th) (tw)} !_c) (a o tw th)>>
        x gets the value (a o)
        c gets the value (th)

<prog [x]
        <is? ({star a} _x) (a a a a)>>
        x gets the value a
```

The argument of the actor WHEN is a list of clauses. If the object that the actor WHEN is trying to match has the property that it matches the first element of one of the clauses then it must match the rest of the elements in that clause.

```
<prog [[!=fix x]]
        <is? <when [<?> _x]> 3>>
        x gets the value 3 since 3 is a fixed point number.
```

In the expression below <all _a _b> matches 3 only if both _a and _b match 3. Thus both a and b are set to 3.

```
        <prog [a b]
                <is? <all _a _b> 3>>
```

A number of actors are defined below.

A palindrome is defined to be a list that reads the same backwards and forwards. Thus (a (b) (b) a), (), and ((a b) (a b)) are palindromes. More formally in MATCHLESS, a palindrome can defined as an actor of no arguments:

```
<define palindrome
        <actor [ ]
                ;"palindrome is a actor of no arguments"
                <either
                        <empty>
                        ;"a palindrome is either empty or"
                        <declaration [x]
                                ;"declare a new local x"
                                <list _x {palindrome} .x>
                                ;"let x be the first element of the
                                        linear structure.
                                        Also x must
                                        be the last element
                                        with a palindrome
                                        in between">>>>
```

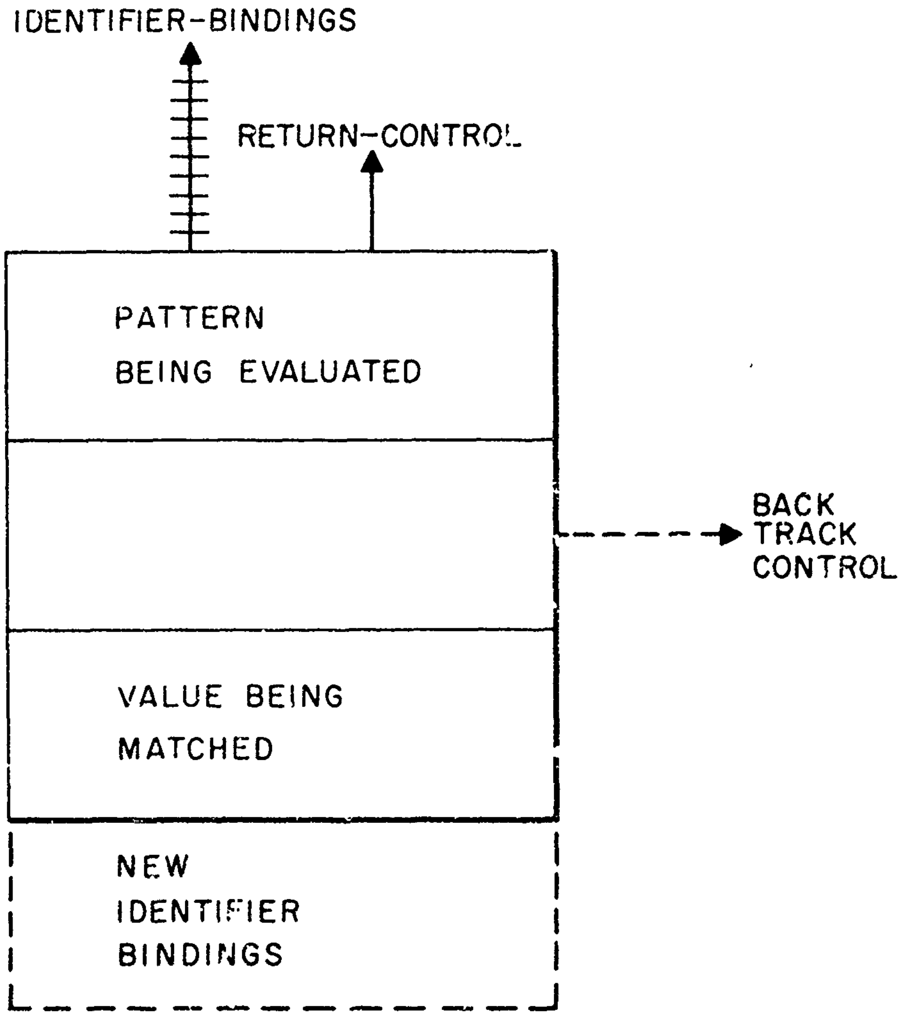
For example

<is? <palindrome> (a 1 1 a)> is true.

The form ACTOR is like the function of LISP except that it is used in actors instead of in functions. The above definition reads: a palindrome is a list or vector such that it is empty or it is a list or vector which begins and ends with x with a palindrome in between. The actor SAME causes the identifier x to be rebound every time that palindrome is called. The actor REVERSE is defined to be such that

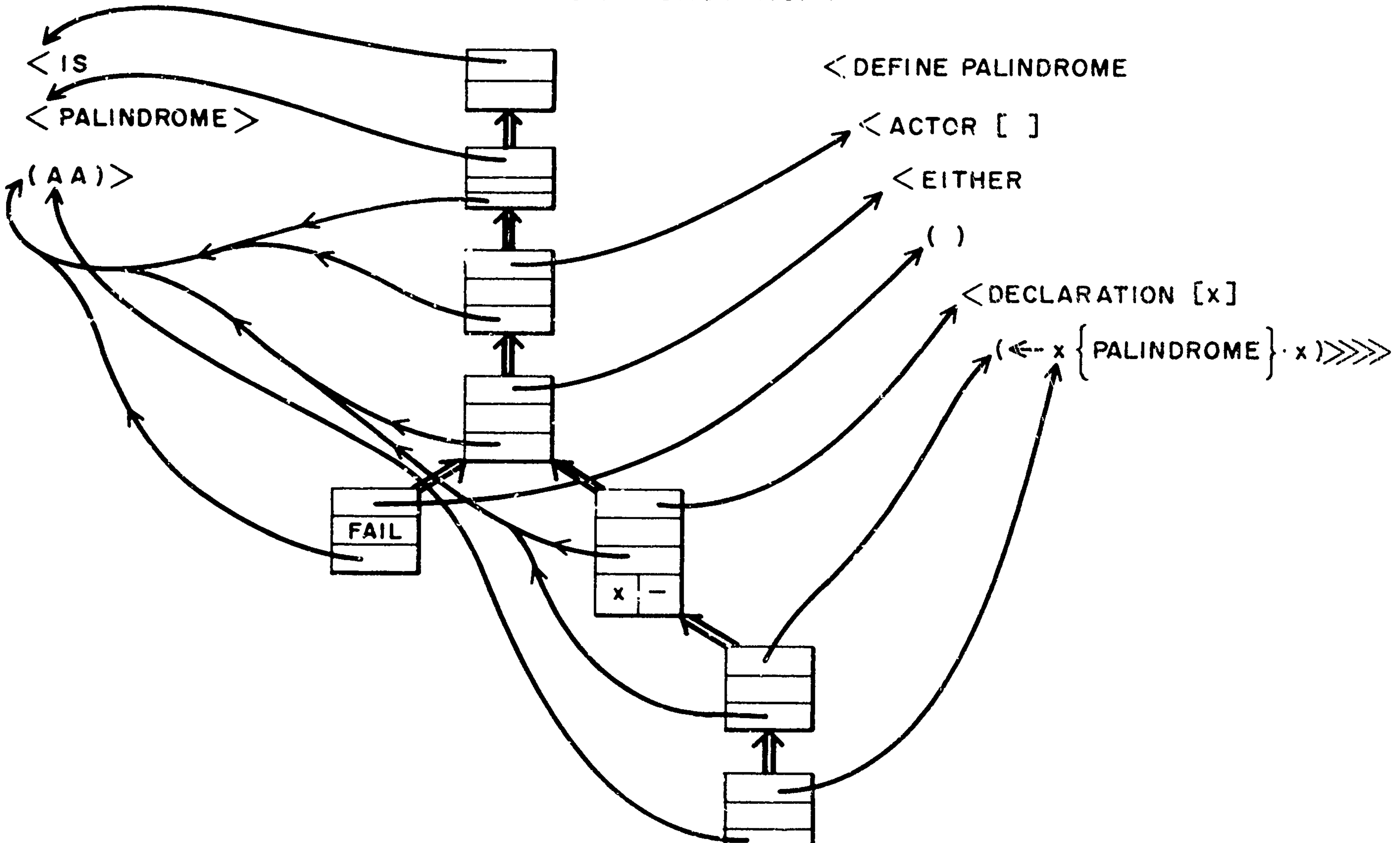
4.6 page 175a

FORMAT OF ACTOR ACTIVATIONS
IN SNAPSHOTS



NOTE : THE IDENTIFIER-BINDINGS AND RETURN-CONTROL POINTERS OF AN ACTIVATION ARE USUALLY THE SAME AND THUS ARE COMBINED INTO A DOUBLE POINTER LIKE THIS.

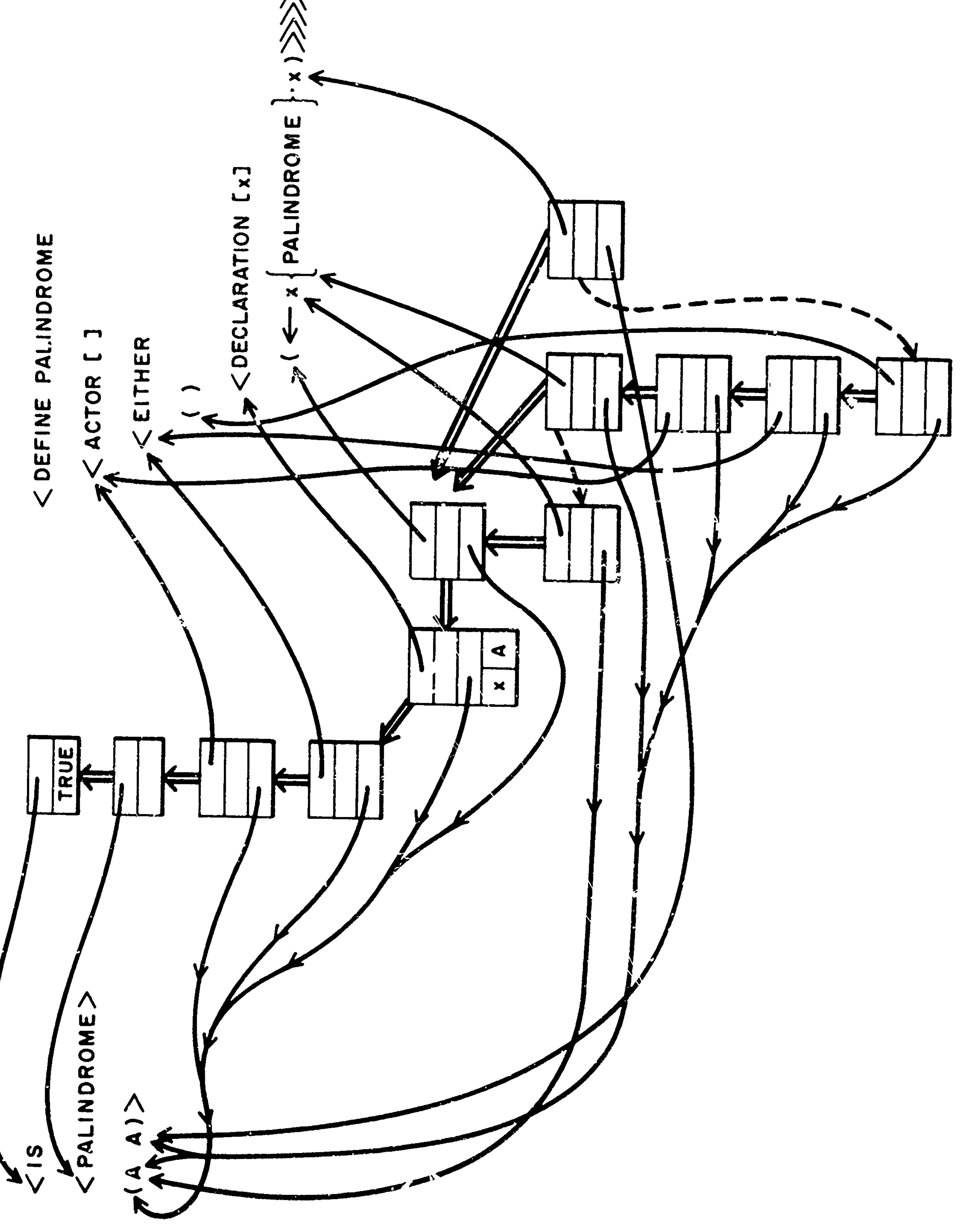
SNAPSHOT NO. 1



4 6 page ~~1~~75b

4.6 page 175c

SNAPSHOT NO. 2



<is? <reverse .x> .y> is true only if the value of x is the reverse of the value of y. The definition of reverse is

```
<define reverse
        <actor [x]
                <when
                    [<monadic>
                        ;"if the object being matched is monadic
                                then it must be equal to x"
                        .x]
                    [<declaration [first rest]
                                ;"otherwise let first
                                        be the first element
                                        of the matching object
                                        and rest
                                        be the segment of the rest of
                                        the elements of the
                                        matching object."
                                <linear _first !_rest>
                                ;"when <linear {reverse .rest} .first>
                                        matches .x we are done"
                                <be <is?
                                        <linear
                                                {reverse .rest}
                                                .first>
                                        .x>>>]>>>
```

For example

<is? <reverse (x y z)> (z y x)> is true

Many of the ideas for the actors come from Post productions, BNF, general regular expressions, ELINST (Slagle's algebraic pattern matcher), SNOBOL, CONVERT, and LISP. We give examples of the use of these actors afterward.

4.6.1 Definitions of Actors

4.6.1.1 Control Actors

4.6.1.1.1 Conditional Actors

<==

|x|> matches an object only if the value of |x| is identical to the object.

<NON

|pattern|> matches an object only if |pattern| does not match the object. Thus <non c> matches a, but <non a> does not match a.

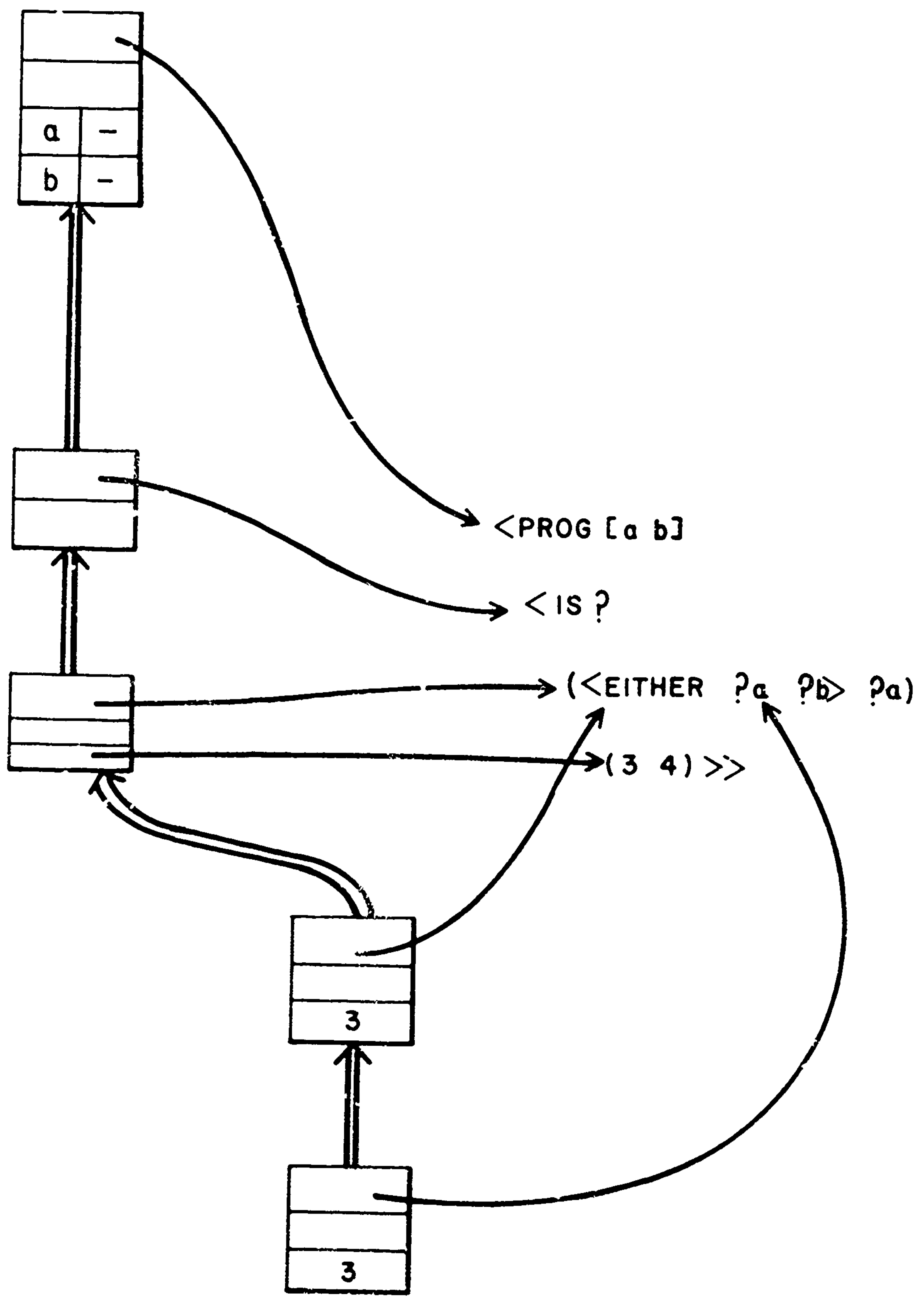
<VEL

-patterns-> matches an object only if some pattern in turn matches the object. If a simple failure backs up to the actor VEL, then the next alternative pattern in turn is tried. If all the alternatives are exhausted, then VEL itself propagates a simple failure backward. For example

```
<prog [[a 3]]
        <_ (<vel 4 _a> <+ .a 1>) (4 5)>>
a is initialized to 3
4 is matched  with 4
<+ 3 1> fails to match 5.
_a is matched against 4 giving a the value 4
<+ 4 1> matches 5

<prog [a b]
        <_
                (<vel ?a ?b> ?a)
                (3 4)>>
a gets the value 3
3 does not match 4 so a simple failure is generated
a gets the value #unassigned
```

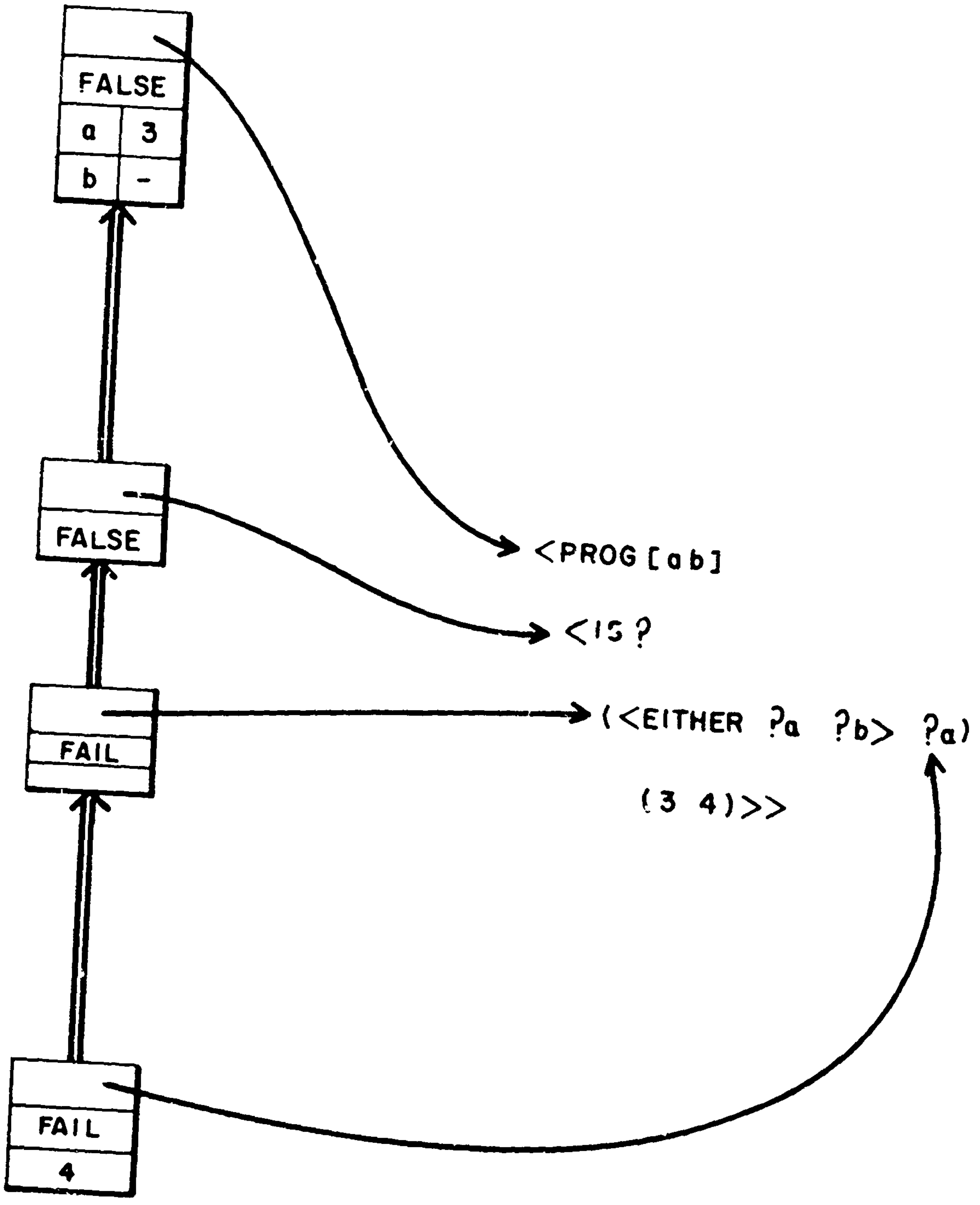
4.6 page 177a

SNAPSHOT NO. 1

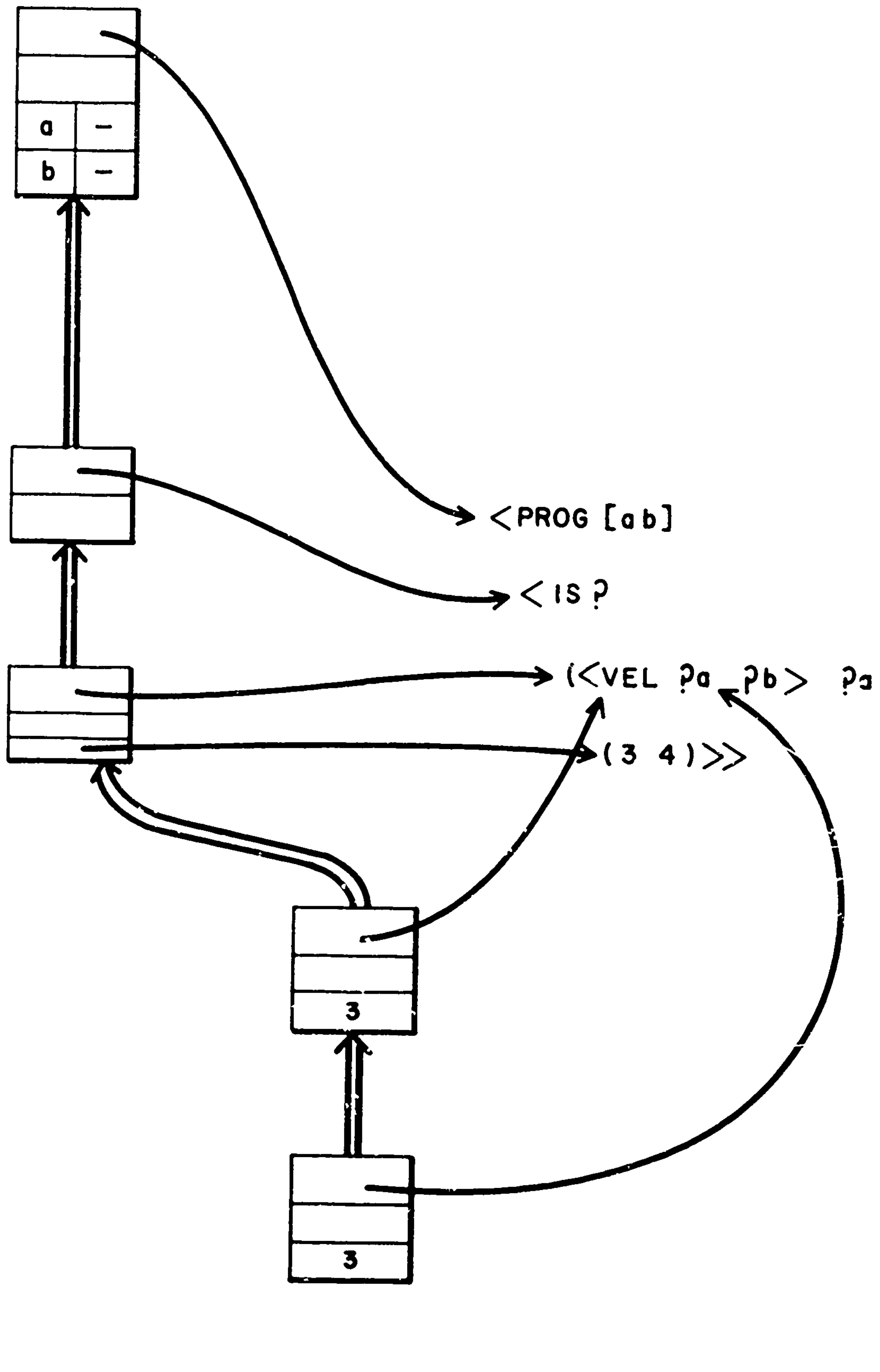


4.6 page 177b

SNAPSHOT NO. 2

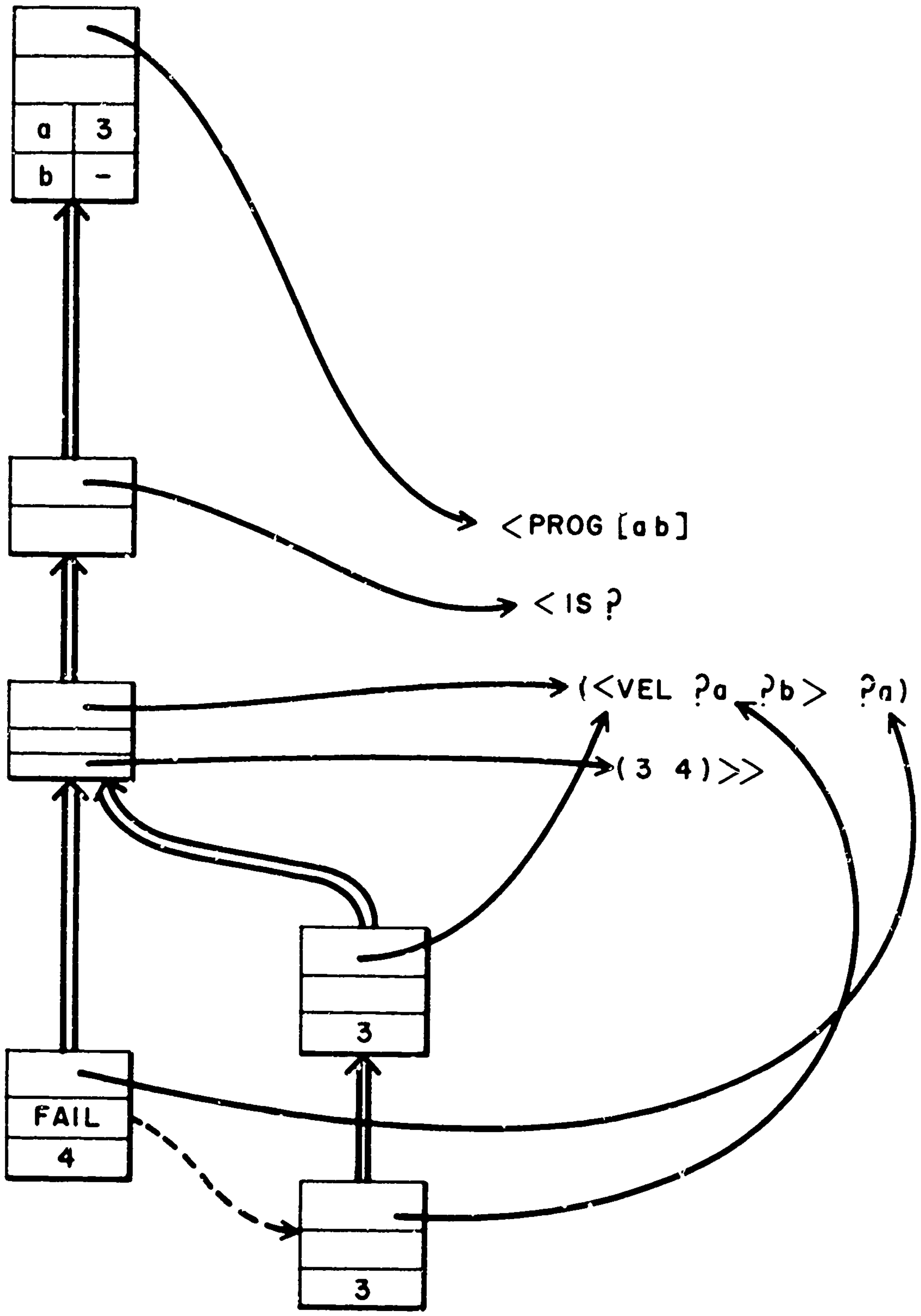


SNAPSHOT NO. 1

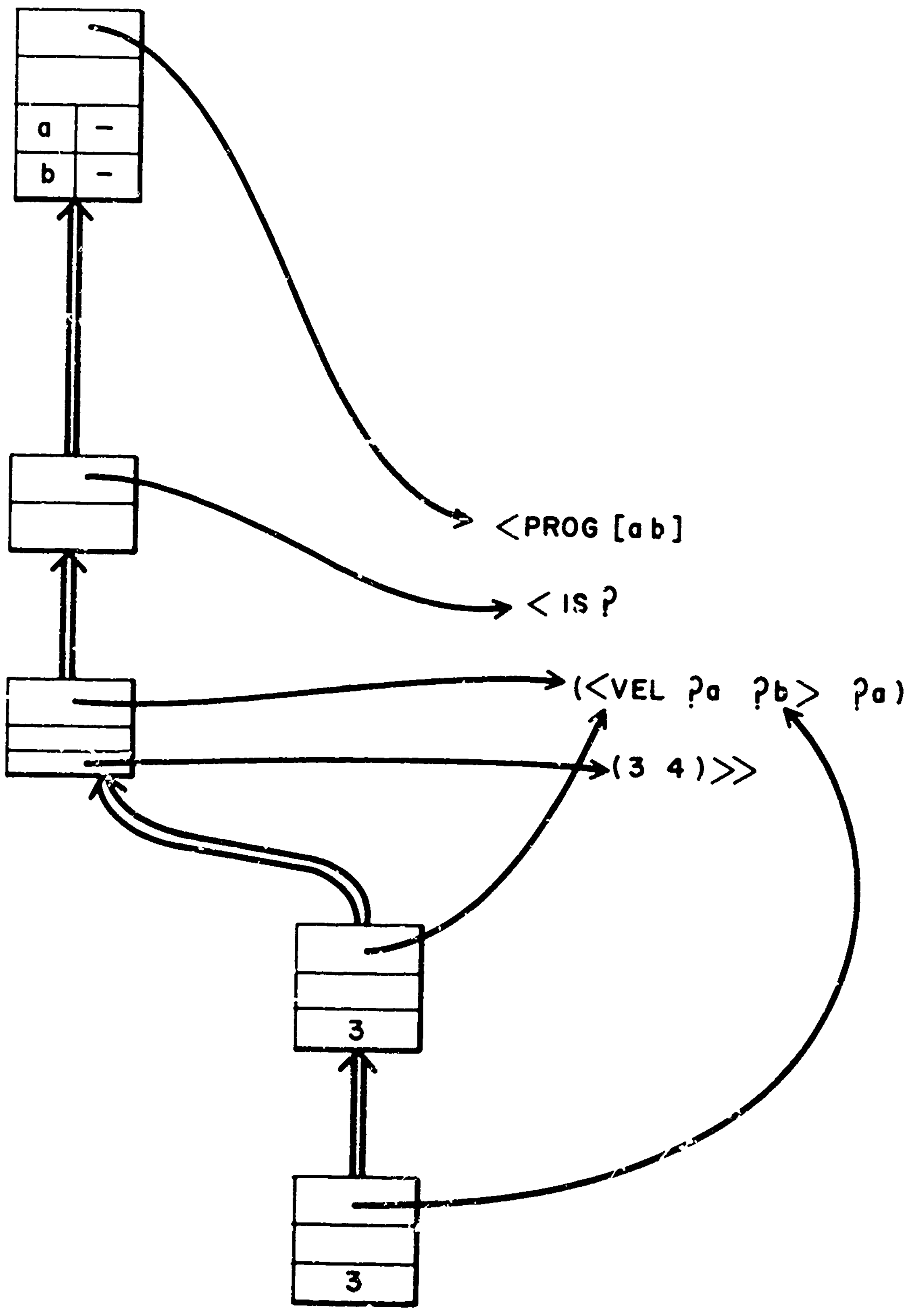


4.6 page 177d

SNAPSHOT NO. 2

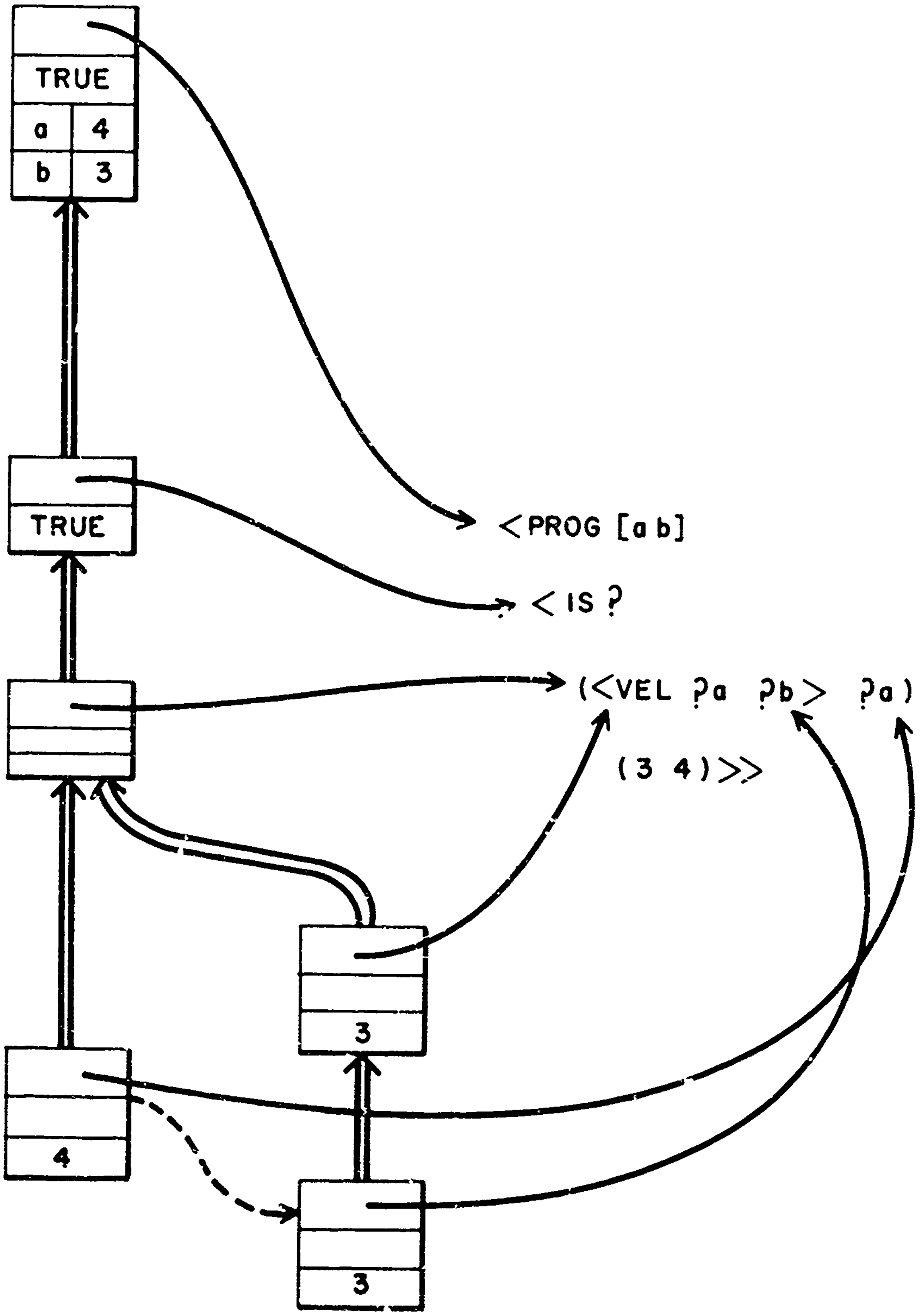


SNAPSHOT NO. 3



4.6 page. 177 F

SNAPSHOT NO. 4



```
b gets the value 3
a gets the value 4
```

The following example shows how VEL is different from EITHER:

```
<prog [a b]
        <is?
                (
                        <either ?a ?b>
                        ?a)
                (3 4)>>
```

evaluates to <> which is false since EITHER does not try matching ?b with 3 because ?a successfully matched 3.

<ALL

-patterns-> matches an object only if each pattern in turn matches the object.

<IS-ACTOR

|pattern|> will match an object only if the object matches the value of |pattern|.

<BE

|predicate|> matches an object only if the |predicate| is not false. In other words the actor BE ignores the object that it is supposed to match and considers only the value of predicate.

<be <is? 3 3>> matches anything

<be <>> does not match anything since <> is false.

<MATCHING

[|object| |tail| |loc|] |predicate|> is exactly like the actor BE except that the identifier |object| is bound to the object being matched, |tail| is bound to its tail if it has one, and |loc| is bound to a locative to |object| if there is one.

<WHEN

+checker+ -clauses-> where each clause is of the form [|pattern| -more-patterns-] or of the form #DECLARE [[-declarations-] |pattern| -more-patterns-] matches an object if the first element of some clause in turn matches the object and then the rest of the elements in that clause match the object.

```
<prog [x y]
      <is?
            <when [<number> _x] [_y]>
            foo>>
;"y gets the value foo since foo is not a number"

<prog [[!=fix [y 1] x]]
      <is?
            <when
                  [<be <is? _x <+ .y 1>>>
                  <+ .x 2>]>
            4>>
;"x gets the value 4"
```

4.6.1.1.2 Block Structuring

<DECLARATION

[-declarations-] -patterns-> matches an object only if each pattern in turn matches the object after -declarations- are bound.

```
<_ #declaration[[x] _x] 4>
x gets the value 4
```

<ACTIVE

<|procedure| -args-> |place|> matches the pattern |procedure| against all the currently active procedures within

|place|. If the match succeeds then -args- are matched against the procedures arguments. The |place| may either be a process or of the form [¬"BETWEEN" |name1| |name2|] where |name1| and |name2| are the names of blocks for a process.

4.6.1.2 Data Actors

4.6.1.2.1 Specialists

4.6.1.2.1.1 Structure Actors

Any expression delimited by "(" and ")" matches a list. Any expression delimited by "[" and "]" matches a vector or a tuple.

<?> matches anything.

<?

|n|> matches an object only if the object has length the value of |n|. For example the following are true:

```
<is? <?> (b a c)> is true.
<is? ({?}) ()> is true.
<is? (a {?}) (a)> is true.
<is? (a {?}) (a b)> is true.
```

Something of the form '|x| matches only those objects which are equal to |x|. For example '.a matches .a and 'a matches a.

<?

|n| |pattern|> will match anything of length |n| which in turn matches |pattern|.

```
<prog [four-characters]
      <is?
            [<? 3> <? 4 :four-elements> <?>]
            [a b c d e f g h i]>
      ;"four-elements has the value [d e f g]">
```

<STAR

-patterns-> matches an object only if the object consists of a sequence (including the null sequence) of elements that match patterns. For example <star 3> matches (3 3 3) and (a {star b c} e) matches (a b c b c e).

<DAGGER

-patterns-> matches an object only if the object consists of at least one sequence of elements that match patterns. For example <dagger 3> matches (3) and (3 3) but does not match ().

<OPTIONS

-patterns-> matches a sequence of elements which match a subsequence of the patterns from left to right. For example <options a !=fix !=atom> matches (a 3).

<HAS

-properties-> matches any object with the appropriate properties where each property is of one of the following forms:

[|indicator| <FAIL>] fails if there is an object under |indicator|.

[|indicator|] removes the value under the |indicator| if it is present

[|indicator| |pat|] says that the object has under the |indicator| a piece of data which matches the pattern |pat|.

The actor HAS allows MATCHLESS to do pattern matching on arbitrary graph structures. The example of the syntax of LISP given below shows how we can write grammars over graphs. The idea of developing pattern structures over graphs has been generalized and extended in PLANNER.

```
                <_
                        <has ["x" 3] [4] [c <replace 5>]>
                        <node [c [4]] [4 5] ["x" 3]>>
```

evaluates to

```
                #node  [["x" 3] [c 5]]
```

```
        <SELECT
```

|pat| |other|> matches any structue such that one of the elements of the structure matches |pat| and the remainder of the structure matches |other|.

```
                <prog [r]
                        <select 3 _r> <class 4 3 5>
                        ;"r gets the value <class 4 5>">
```

```
        <OF
```

|pat| |collect| |other|> matches any structure such that the list of all the elements of the structure that match |pat| matches the pattern |collect| and the rest match the pattern |other|. For example

```
                <prog [integers others]
                        <_
                                <of !=fix _integers _others>
                                [a 3 b 5 9]>>
                        integers gets the value (3 5 9)
                        others gets the value (a b)
```

```
        <STRUCTURE>
```

matches any list, vector, or node.

<EMPTY>

matches any empty structure.

<MONAD>

matches any object which cannot be decomposed.

<LINEAR

-patterns-> matches any list, vector, or tuple whose elements match the patterns in order. For example <linear 3 4> matches (3 4) and it also matches [3 4].

<ELEMENT

|x|> matches any object such that the object is an element of |x|.

<CONTAINS

|pat|> matches any structure which contains an object that matches |pat|.

```
!%<block (<oblist contains!-> <oblist>)>

<define contains
        <actor
                ['y]
                <container
                        <eval !'<actor () .y>>>>>

<define container <actor [x]
        <when
           [<.x>
                ;"if the actor x matches
                        the matching object
                        then we are done"]
           [<monadic>
                ;"if the matching object is
                        monadic then fail"
                <fail>]
           [<linear <container .x> .{?}>
```

```
                                ;"if the first element in the
                                      matching object contains x
                                      then we are done"]
                           [<?>
                                ;"else the rest of the matching object
                                      must contain x"
                                <linear <?> {container .x}>]>>>

                   !%<end-block>
```

<REPLACE

|x|> matches any object. As a side effect the object which is matched is replaced with the value of |x|.

```
                   <prog [y]
                         <is?
                               <all
                                     _y
                                     (<replace a> {replace (b)})>
                               (c d e)>>
                   y gets the value (a b)
```

We can define an actor rev which changes any list which it matches to the reverse of that list.

```
                   <define rev <actor [ ]
                         <either
                               <empty>
                               <linear <?>>
                               <declaration [first last]
                                     <linear :first {?} :last>
                                     <linear
                                           <replace .last>
                                           {rev}
                                           <replace .first>>>>>>
```

Now if evaluate

```
                   <_ :c (e f g)>
                   <_ <rev> .c>
```

then c mysteriously has the value (g f e) because the actor rev destroys the initial list to make the reverse.

<PRECEDES

|x|> will match any expression which precedes |x| in the total ordering on expressions. For example <precedes "c"> will match "a" since "a" prcedes "c".

<FOLLOWS

|x|> will match any expression which follows |x| in the total ordering on expressions.

4.6.1.2.1.1.1 List

<LIST!-DECOMPOSER

-patterns-> matches lists whose elements match -patterns-. It is equivalent to (-patterns-).

4.6.1.2.1.1.2 Vector

<VECTOR!-DECOMPOSER

-patterns-> matches vectors whose elements match -patterns-. It is equivalent to [-patterns-].

4.6.1.2.1.1.3 String

<STRING!-DECOMPOSER

-patterns-> matches strings whose substrings match -patterns-.

```
        <prog [first rest]
              <_
                  <string !_first " " !_rest>
```

```
                    "see the boy">>
                first gets the value "see"
                rest gets the value "the boy"

            <prog [root]
                <_ <string !_root "s"> "cats">>
            root gets the value "cat"
```

4.6.1.2.1.1.4 Graph

<NODE!-DECOPOSER

-properties-> is equivalent to <ALL !=NODE <HAS -properties->>.

4.6.1.2.1.2 Atom

<ATOM!-DECOMPOSER

|s| |o|> will match an atom whose print name is the string |s| and which is on the oblist named |o|.

4.6.1.2.1.3 Word and Number Actors

<NUMBER > matches an object only if the object is a number. For example <number> matches 3.

<LESS

|n|> matches any number less than the value of |n|.

<LESS=

|n|> matches any number less than or equal to the value of |n|.

<GREATER

|n|> matches any number greater than the value of |n|.

<GREATER=

|n|> matches any number greater than or equal to the value of |n|.

<FIELDS

-specifications-> matches any fixed point number which meet each specification of a field in turn. A fixed point number x meets a specification of the form [|bits| |pattern|] only if the number which is the byte of x defined by |bits| matches |pattern|. The expression <bits |s| |p|> defines a byte |s| bits wide which is |p| bits from the right end of the word.

<fields [<bits 3 0> 4] [<bits 1 35> 1]> matches a fixed point number whose lower 3 bits are 4 and whose sign bit is on.

4.6.1.2.1.4 Algebraic Actors

The motivation for providing algebraic actors is to enable pattern directed algebraic simplification to be easily accomplished. Often it is not clear which simplified form is most useful. Using the hierarchical backtrack control structure of PLANNER one form can be tried as a hypothesis and then in the light of this experience perhaps another more suitable one.

<!+

-patterns- |rest-of-summands|> matches a sum such that each pattern matches a summand and the rest of the summands match the pattern |rest-of-summands|.

```
               <is?
                       <!+ a b <?>>
                       '<+ c b a>> is true.

               <prog [y z x]
                       <is?
                               <!+ <all <non c> _z> _y _x>
                               '<+ c b a>>>
               z gets the value b
               y gets the value c
               x gets the value a

               <prog [y x]
                       <is?
                               <!+ _y b _x>
                               '<+ 1 c b d>>>
               y gets the value 1
               x gets the value <+ d c>
```

<SUM-OF

|pat| |terms-that-match-pat| |rest-of-summands|> matches any sum such that the sum of the summands that match |pat| in turn match the pattern |terms-that-match-pat| and the rest of the summands match the pattern |rest-of-summands|.

```
               <prog [y]
                       <is?
                               <sum-of <!* x <?>> _y <?>>
                               '<+ <* 3 x> <* y a>>>>
               y gets the value <+ <* 3 x>>
```

<!*

-patterns- |rest-of-factors|> matches a product of factors such that each pattern matches a factor in the product and the rest of the factors match the pattern |rest-of-factors|.

```
               <is? <!* 5 b c> '<* c b 5>> is true.
```

```
<prog [x y]
        <is?
                <!* <all <number> _x>  _y <?>>
                '<* <+ 2 a> 3 a>>>
x gets the value 3
y gets the value <+ 2 a>

<prog [x]
        <is? <!* 3 _x 1>  0>>
x gets the value 0
```

<PRODUCT-OF

|pat| |factors-that-match-pat| |rest-of-factors|>

matches any product of factors such that the product of the factors that match pat in turn match |factors-that-match-pat| and the rest of the factors match the pattern |re. .-of-factors|.

```
<prog [x y]
        <is?
                <product-of <non <number>> _x _y>
                !'<* a 3 b 5.0>>>
x gets the value <* a b>
y gets the value <* 3 5.0>
```

<POWER

|base| |exponent|> matches an exponential.

```
<prog [x y]
        <is? <power _x _y> '<expt y 2>>>
x gets the value y
y gets the value 2

<prog [x y]
        <is? <power _x _y> 0>>
x gets the value 0
```

<EXTRACT

|pat| |terms-with-pat-extracted| |rest-of-terms|>

matches a sum of terms such that the sum of the terms which contain a factor which matches |pat| matches |terms-with-pat-extracted| and the sum of the rest of the terms matches the pattern |rest-of-terms|. The actor EXTRACT is due to W. Bledsoe.

```
        <is?
                <extract x <!+ 3 a 0> y>
                !'<+ !'<* a x> y !'<* x 3>>> is true
```

Joel Moses invented the example of defining a quadratic in x using patterns.

```
<define quadratic
  <actor
        [x a b c]
        <extract
                <power .x 2>
                <all <non 0> <non <contains .x>> <.a>>>
                <extract
                        .x
                        <all <non <contains .x>> <.b>>>
                        <all <non <contains .x>> <.c>>>>
```

```
Thus if
        <prog [["special" a1 b1 c1]]
                <is?
                        <quadratic
                                y
                                <actor [ ] _a1>
                                <actor [ ] _b1>
                                <actor [ ] _c1>>
                        <!+
                                a
                                <!* 3 y>
                                <!* z '<expt y 2> 4>
                                <!* c y>>>>
```

```
then
        a1 gets the value <* z 4>
        b1 gets the value <+ 3 c>
        c1 gets the value a
```

4.6.1.2.1.5 Locative

4.6.1.2.2 Type

<OF-TYPE

|atom|> matches an object |o| only if <==? <TYPE |o|> |atom|> i.e. only if |o| is of the type |atom|. The expression <OF-TYPE |atom|> may be abbreviated as ::|atom|.

<AS

|pat| |inj|> matches an object x only if x is of the type of the range of the injection |inj| and <RETRACT x> matches the pattern |pat|.

4.6.1.3 Identifier

<GIVEN

|theta| -bindings-> acts like <VALUE |theta| -bindings-> if the identifier |theta' has a value. Otherwise <GIVEN |theta| -bindings-> matches an object x only if the identifier |theta| matches x.

?|theta| is an abbreviation for <GIVEN |theta|>

!?|theta| is an abbreviation for {GIVEN |theta|}

<ALTER!-PERSISTENT

|theta| -bindings-> matches any expression x which matches the identifier |theta| and gives |theta| the value x.

:|theta| is an abbreviation for <ALTER |theta|>

!:|theta| is an abbreviation for {ALTER |theta|}

<ALTER!-TENTATIVE

|theta| -bindings-> matches any expression x which can match the identifier |theta|. The identifier |theta| is given the value x. However, if a failure backtrac s to ALTER!-TENTATIVE, then |theta| is restored to its previous value.

_|theta| is an abbreviation for <ALTER!-TENTATIVE |theta|>

!_|theta| is an abbreviation for {ALTER!-TENTATIVE |theta|}

4.6.2 Examples of the Use of Actors

The rest of our examples of the use of actors come from giving a rigorous definition of the syntax of LISP in MATCHLESS. Those readers who are not interested in the details of the syntax of LISP should not read section 4.6.2 The following grammar accounts for essentially all the context dependent features of the LISP syntax. It specifies that a function call must have the right number of arguments. An explicit go must have a tag to which it can go. The syntax specifies that some identifiers are free and others are bound.

```
<define  top-function <actor [ ]
    <declaration
        [["special" [tags () ] [boundvars () ]]]
        (function <varlist> <form>)>>>
```

Thus for example <top-function> matches (function () ()). The actor top-function introduces the pattern identifiers tags and boundvars and binds them to - - which is the null segment.

```
 <define varlist <actor [ ]
        <star
            <declaration
                [[!=atom curvar]]
                _curvar
                <be <is? _boundvars (.curvar !.boundvars)>>>>>>
```

The actor varlist checks each identifier in turn to make sure that it is an atom and then puts the identifier in boundavars.

```
 <define
    form <actor [ ]
    <when
        [<monadic>
            <either <constant> <var>>]
        [
            (!=atom (?))
            <when
                [(prog (?))
                    <progform>]
                [(cond (?))
                    <condform>]
                [(setq (?))
                    (<?> <var> <form>)]
                [(go (?))
                    <goform>]
                [(<has [subr <?>]> (?))
                    (<?> (star <form>))]
                [(<has [expr <?>]> (?))
                  <exprform>]
                [(<has [fexpr <?>]> (?))
                  <?>]
                [(<has [fsubr <?>]> (?))
                    <?>]
                [(<has [lsubr <?>]> (?))
                    (<?> (star <form>))]
                [(<has [lexpr <?>]> (?))
                    (<?> (star <form>))]
                [<?>
                    <matching
                                [expr ¬"optional"]
```

```
                                      <print (.expr undefined)>>]>]
        [((function {?}) {?})
            <function-function>]
        [<?>
            (<form> {star <form>}) ]>>>
```

The above definition says that if a form is a monad then it must be a constant or an identifier; if its first element is an atom then if it begins with the atom prog, then it must be a progform etc.; if it begins with "((function ..) ..)" then it must be a function-function; otherwise it must be a form followed by a formlist.

```
 <define constant <actor [ ] <either t () <number>>>>
```

The only constants are t, (), and numbers.

```
<define var <actor !=atom [ ]
        <either
            <element .boundvars>
            <unbound>>>>
```

An identifier is either in boundvars or it is unbound.

```
 <define condform <actor [ ] (cond {dagger ({star <form>})})>>
 <define
    progform <actor [ ]
    <declaration
        [
           [ "special"
                [tags .tags]
                [localtags () ]
                [boundvars .boundvars]]]
        (prog
                <varlist>
            {all
                <collect-tags>
                <be <is? _tags (!.localtags !.tags)>>
                <star <either !=atom <form>>>})>>>
```

On entrance to progform tags and boundvars are rebound to their previous values. The prog identifiers of the prog are put in boundvars, the tags in the prog are put in tags by collect-tags, and the body of the prog is checked to see if it is well formed.

```
<define
   collect-tags <actor [ ]
<star
     <either
              <declaration [
                             [!=atom curtag]
                             ["special" localtags]]
                         _curtag
                         <be <is?
                                  _localtags
                                  (.curtag !.localtags)>>
                       <when [<element .localtags>
                             <error "multiple defined tag">]
               <?>>>>>>>
```

```
<define
   exprform <actor [ ]
   <declaration
       [args functionvar]
            (
                <has [expr (function _functionvar (?))]>
                (all <star <form>> _args})
            <be <==? <length .functionvar> <length .args>>>>>>
```

An exprform is a call to an expr with the correct number of arguments. Note that immediately inside the actor exprform the identifierss args and functionvar are rebound but remain unassigned.

```
<define goform <actor [ ]
   (go
        <when
             [!=atom
                           <either
                                   <element .tags>
                                   <print (.curtag undefined tag)>>]
             [<form>]>)>>
```

A goform is either an explicit call to go to a tag which must be in .tags or a computed go.

```
 <define
    function-function <actor [ ]
    <declaration
        [args functionvar]
            (
                <declaration
                    [
                        ["special" [boundvars (.boundvars) ]]]
                    (function
                        <all <varlist> _functionvar>
                        <form>) >
                (all <star <form>> _args))
            <be <==? <length .functionvar .args>>>>>>
```

In a function-function the bound identifiers of the function must be added to boundvars and the function-function must have the proper number of arguments.

The above syntax could easily be extended in several directions. For example we could easily modify it so that it would accept type declarations and do type checking. The syntax of MATCHLESS could easily be defined in MATCHLESS.

4.7 HUMBLE

Section 4.7 is logically completely separate from the rest of this report. It is not necessary to read this section to understand the rest of the document.

We are interested in exploring good ways to implement systems like PLANNER on machines. One way is to embed the system in a language like LISP or PL-1. The problem with embedding is that the host language has its own conventions for calling sequences and saving temporaries. The conventions might not be compatible with the system which is being implemented. Another approach is to try to develop a formalism which is sufficiently flexible so that it can adapt to the higher level system conventions but still is efficient enough so that it is feasible to use as an implementation language. The applicative sublanguage of MATCHLESS seems to be at approximately the right level with the restriction that the data are no longer have types associated with them at run time. Thus all the type information must be able to be processed at compile time. The general type definition formalism remains although the definitions must be processed at compile time.

4.8 The Editor

<EDIT

|x|> enables editing the structure |x|. The editor maintains a special identifier CURSOR!-EDIT which represents the position of the editor within the structure. A command may be abbreviated by the first letter in its name. The editor makes use of the tentative versions of the structure modifying commands so that the results of a series of edits can be undone by backtracking. Gregory Phister made suggestions and implemented an editor.

<BENEATH |cursor|> is the expression beneath |cursor| or <> if there is none.

<CONTAINS |cursor|> is the structure which contains |cursor| or is <> if there is none.

<ARC |cursor|> is the indicator under which <BENEATH |cursor|> is found under <CONTAINS |cursor|> or is <> if <CONTAINS |cursor|> is <>. That is if <CONTAINS |cursor|> is not <> then:

```
        <get
                <contains |cursor|>
                <arc |cursor|>> is <beneath |cursor|>
```

<GO |n| |cursor|> moves |cursor| |n| positions to the right if |n| is positive and |n| positions to the left if |n| is negative.

<WALK |n| |cursor|> walks |cursor| |n| positions around the tree.

<UP |n| |cursor|> rises through |n| levels of structure from |cursor|.

<DOWN |n| |cursor|> descends through |n| levels of structure

from |cursor|. If |n| is positive the cursor is moved down to the right otherwise to the left.

<SEARCH |pattern| |n| |cursor|> searches for the |n|th occurence of an object that matches |pattern|. If |n| is positive the search is to the right, otherwise to the left.

<FIND |pattern| |n| |cursor|> will conduct the search only in the object under |cursor|.

<REPLACE |pattern| |x| |n| |cursor|> replaces |n| occurences of objects that match |pattern| with the value of |x|. If |n| is positive the search is to the right, otherwise to the left.

<CHANGE |pattern| |x| |n| |cursor|> changes |n| occurences of objects that match |pattern| with the value of |x| on the structure which is under |cursor|.

<INSERT -expressions- |cursor|> inserts -expressions- into the structure.

<KILL |n| |cursor|> deletes the expression under the cursor and <- |n| 1> expressions following it.

5. PLANNER

The PLANNER formalism incorporates a unified set of problem solving primitives that run under a multiprocess backtrack control structure. The formalism itself is independent of any particular problem solving domain. The primitives of the formalism make default decisions in the course of a computation in those cases where the information supplied does not specify exactly what is to be done. However, as a matter of principle each primitive allows a continuum of expression from no preference at all down to the specification of exactly one choice. The formalism is intended to be used as a matrix in which the necessary domain dependent knowledge can be embedded. Many of the primitives rely or side effects to accomplish their purpose. Although the use of side effects is in opposition to some theories of good language design, their use in PLANNER has worked out well. The formalism encourages modular programming through the use of specialized routines to satisfy goals and make deductions.

The name PLANNER comes from the desire to create a formalism in which it is easy to express plans of action. To construct a plan in the formalism is the same as constructing a PLANNER theorem. Mixing planning and deduction is quite easy. Conditional plans are explicitly provided for as is the ability to backtrack in case of failure.

Consider a statement that matches the pattern [IMPLIES |x|

|y|]. The statement has several imperative uses.

st1: If we can deduce |x|, then we can deduce |y|.

In PLANNER the statement st1 would be expressed as <ANTECEDENT [] |x| <ASSERT |y|>> which means that |x| is declared to be the antecedent of a theorem such that if |x| is ever asserted in such a way as to allow the theorem to become activated then |y| is asserted.

```
st2: if we want to deduce |y|,
             then establish a subgoal to first deduce |x|.
```

In PLANNER the statement st2 would be expressed as

```
<CONSEQUENT [] |y|
        <GOAL |x|>
        <ASSERT |y|>>
```

which means that |y| is declared to be the consequent of a theorem such that if the subgoal |x| can be established using any theorem then the consequent |y| is asserted.

We could also assert <CLAUSE [] [NOT |x|] |y|> which is a clause which says that [not |x|] or |y| is the case. PLANNER has goal oriented primitives for using and manipulating all of the above variants. For certain purposes any one of the variants can be more useful than the others. Imperative information and heuristics can more easily be expressed in the procedural variants. For example heuristic information as to when we should create a subgoal x in order to achieve y can more easily be incorporated into a CONSEQUENT theorem. [On the other hand we can more easily deduce <CLAUSE []-c

d> from <CLAUSE [ ] x [NOT y] c d>] Of course the distinction is not sharp since the two kinds of assertions can be combined by making assertions about the actions of imperatives.

5.1 PLANNER Forms

5.1.1 Hierarchical Backtrack Control Structure

PLANNER uses a control structure in which the hierarchy of calls is preserved so that a computation can backtrack to an activation from which it has already returned. Backtracking preserves the nesting of block structure. It simply traverses the statements executed in reverse order. The primitive functions FAIL and FAILPOINT enable the backtrack process to be controlled. The form <FAIL> generates a simple failure which backtracks to the most recently executed form

```
<FAILPOINT +activation-name+ [-declarations-]
        |expression|
        [|message| |activation|]
        -body->
```

Where |message| is bound to the message of the failure and the predicates are evaluated to try to find one which is true. For example

```
<prog [[x 3]]
       <+
                <prog foo []
                        <failpoint []
                                .x
                                [¬"optional"]
                                <.foo <_ :x 4>>
                                ;"exit .foo with 4">>
```

```
                    ;"the first time through the above expression
                         has .x as its value"
                    <cond
                       [<is? 3 .x>
                          <fail>]
                       [¬"else"
                          5]>>>
```

evaluates to <+ 4 5> which is 9.

The identifier x is declared to be a fixed point integer which is initialized to 3. The value of <failpoint [] .x [¬"optional"] <_ :x 4>> is 3. When the second argument of the call to "+" is evaluted the conditional detects that x is bound to 3 and so generates a simple failure. The failure backtracks to the call to FAILPOINT with the message <> which is FALSE. The identifier x is assigned the value 4 and the rest of the computation proceeds normally.

The top level function of PLANNER is a read, evaluate, print loop. When the expression read is successfully evaluated then the whole hierarchy of calls is forgotten, the value is printed, and the process repeats.

One of the most straight forward ways to implement hierarchical backtrack control structure is through the use of a backtrack stack on which backtrack information is stored. The only tricky point comes in the executon of an exit where the temporaries must be pushed onto the backtrack stack before doing the exit. The other straight forward method of implementation is not to have a stack at all but rather to keep all the activation frames in garbage collected storage. The stack implementation has the advantages that it keeps a smaller working set and doesn't cause garbage collection.

The swamp implementation has the advantages that it is conceptually cleaner and is more flexible. The ideal implementation is to be able to run either mode. In stack mode the acivation records are simply tuples on the stack.

The use of backtrack control structure has the important fringe benefit that it allows us to debug more easily. We have available the following control prmitives.

<STEP |p| |n| |condition|> executes the process |p| for |n| elementary steps unless the |condition| is met in which case it returns the number of elementary steps completed. If |n| is negative then the process is executed BACKWARDS! This enables us to zero in on bugs by running forwards and backwards until the bug is found.

<INVOKE |p| |n| |condition|> executes the process |p| for |n| procedure invocations unless the |condition| is met in which case it stops and returns the number of procedural invocations which have been completed. Again if |n| is negative then the process is run backwards.

5.1.2 PLANNER Functional Forms

The functional forms in PLANNER are FUNCTION and ACTOR. The sole change in the semantics is that the functional forms of PLANNER can handle pattern directed invocations.

The following example illustrates the syntax of functional forms. The function AMONG which is defined below is a generally useful PLANNER function. What AMONG does is to successively return the elements of the structure given as its argument. For example <among [E A]> returns E as its value. But if a simple failure backtracks to it then it returns A as its value and continues the computation. But if still another simple failure backtracks then it allows the failure to continue to propagate through the function AMONG. The particular way in which the function AMONG is used here does not accomplish anything that cannot be done easily in LISP. We give this example because it is simple enough to be easily understood. One way to assign to the identifier x the value which is the first element of .list that is greater than 5 would be

```
        <is
                ({?} <all <greater 5> :x> {?})
                .list>
```

Another way would be <is _x <larger 5 <among .list>>> where

```
<define among <function munger [list] <prog [first]
        <failpoint forward []
           <>
                ;"establish a failpoint and return <>"
            [m a?]
                ;"on backtracking let m be the message and a? be true
if
                        the failure will propagate through"
                <cond
                    [<not? <is? .m <>>>
                        ;"if the message is not <>
                                then restart the failure"]
                    [<is? .m <>>
                        ;"the message is <>"
                        <restore .forward>
                        ;"start going forward agai
                                with the failpoint restored"]>>
        <cond
            [<empty? .list>
```

```
                    ;"if list is empty generate a
                            simple failure out of among"
                    <fail <> .munger>]
                [¬"else"
                    <_ <linear :first !:list> .list>
                    ;"set first to the first of .list and
                            list to the rest of .list"
                    <.munger .first>
                    ;"exit .munger with .first"]>>>>

<define larger <function [a b]
        <cond
            [<is? <greater .b> .a>
                ;"if a is greater than b then return a"
                .a]
           [¬"else"
                ;"otherwise generate a failure with the message <>"
                <fail <>>]>>>
```

Thus the value of <larger <among (2 4 6)> 5> is 6.

5.1.3 PLANNER Theorems

PLANNER allows procedures to be invoked by a pattern which states what the procedure is supposed to accomplish.

There are four kinds of theorems which are presently defined in the language for satisfying requests made in the body of procedures:

1. Consequent theorems for satisfying goals. Consequent theorems are the most fundamental in the sense that they can easily be used to simulate the other two kinds of theorems.

2. Antecedent theorems for deducing the conclusions of

PATTERN DIRECTED INVOCATION
DATA BASE OPERATIONS

	INTERROGATION	INSERTION	DELETION
KIND OF THEOREM	<CONSEQUENT [-DECLARATIONS-] PAT 1 -BODY->	<ANTECEDENT [-DECLARATIONS-] PAT 1 -BODY->	<ERASING [-DECLARATIONS-] PAT 1 -BODY->
PATTERN DIRECTED INVOCATION	<[GOAL PAT 2]>	<[ASSERT PAT 2]>	<[ERASE PAT 2]>
PATTERN DIRECTED INVOCATION TRIGGERED BY DATA BASE OPERATION	<GOAL PAT 2>	<ASSERT PAT 2>	<ERASE PAT 2>

NOTE: IN ALL CASES PAT 1 MUST MATCH PAT 2 IN ORDER FOR THE INVOCATION TO BE SUCCESSFULL.

assertions

3. Erasing theorems for deducing conclusions from the fact that some assertion is no longer true

4. Simplifying theorems are for simplyfying expressions.

5.1.3.1 Consequent

```
        <CONSEQUENT
                -type- +activation-name+
                |declaration-specification|
                |consequent-pattern|
                -body->
```

evaluates to a procedure which declares that |consequent-pattern| is the consequent of a theorem which can be used to try to establish goals that match the pattern |consequent-pattern|. Whether or not the theorem actually succeeds in establishing the goal depends on the body. Typically the first action that a theorem of type consequent takes is to try to reject the goal. We cannot emphasize too strongly the importance of analyzing the consequences of goals in order to reject the ones which cannot be achieved. Even if no absurdity is detected, the consequences are often just the statements that are needed to establish the goal. The only way that a theorem that begins with the atom consequent can be called is by the pattern directed call:

```
                <CALL
                        [<[GOAL |goal-pattern|]>
                                |recommendation|
                                |state-path|]>
```

which attempts to satisfy the goal |goal-pattern| where |consequent-pattern| matches |goal-pattern| and the consequent theorem is in the data base specified by |state-path|. The function CONSEQUENT is defined to be:

```
        <FUNCTION +checker+ +activation-name+
                [¬"PATTERN"
                        [|declarations| [GOAL |consequent-pattern|]]]
                -body->
```

The following theorem says that if it is our goal to prove x and we have proved that w implies x then we should make it our goal to prove w.

```
        <consequent [x w] ?x
                <current [implies ?w ?x]>
                <goal .w>>
```

The following theorem says that two things are equal if they are identical.

```
        <consequent [x] [= ?x ?x]>
```

With this consequent theorem, evaluating the following causes:

```
        <prog [a]
                ;"declare an identifier a"
                <goal [= ?a 3]>
                ;"a gets the value 3 since a is linked to the
                        identifier x in the consequent theorem">

        <prog [a c]
                ;"declare a and c"
                <prog [b]
                        ;"declare b"
                        <goal [= ?a ?b]>
                ;"a is linked to b"
                <goal [= ?b ?c]>
                ;"b is linked to c">
        <goal [= ?A 3]>
        ;"a gets the value 3 and so
                therefore c gets the value 3">
```

5.1.3.2 Antecedent

```
        <ANTECEDENT
                +checker+
                |declaration-specification|
                |antecedent-pattern|
                -body->
```

evaluates to a theorem which declares that |antecedent-pattern| is the antecedent of a theorem from which conclusions may be drawn by the body. The theorem can be used to try to deduce consequences from the fact that a statement that matches the antecedent has been asserted. The only way that a theorem that begins with the atom antecedent can be called is by the pattern directed call:

```
                <CALL
                        [<[ASSERT |assert-pattern| ]>
                                |recommendation|
                                |state-path| ]>
```

which draws conclusions from |assert-pattern| where |assert-pattern| matches |antecedent-pattern| the antecedndent theorem statisfies |recommendation| and the antecedent theoren is in the data base specified by |state-path|. The function ANTECEDENT is defined to be:

```
        <FUNCTION +checker+ +activation-name+
                [¬"PATTERN"
                        [|declarations| [ASSERT |antecedent-
pattern| ]]]
                -body->
```

The following theorem says that if we assert something of the form [not [implies X Y]] then we should deduce X.

```
        <antecedent [x y] [not [implies _x _y]] <assert .x>>
```

The following theorem says that if something of the form [marry |x| |y|] is asserted then [bachelor |x|] should be erased.

```
        <antecedent [x y]
                [marry _x _y]
                <erase [bachelor .x]>>
```

5.1.3.3 Erasing

```
        <ERASING
                -type-
                |declaration-specification|
                |erasing-pattern|
                -body->
```

can be used to try to deduce consequences from the fact that a statement that matches the pattern |erasing-pattern| has been erased. The only way that a function of kind erasing can be called is by the expression

```
                <CALL
                        [<[ERASE |erase-pattern|]>
                                |recommendation|
                                |state-path|]>
```

which expresses the fact that there has been a change in the world affecting |erase-pattern| where |erase-pattern| matches |erasing-pattern|. The function ERASING is defined to be:

```
        <FUNCTION +checker+ +activation-name+
                [-"PATTERN"
                        [|declarations| [ERASE |erasing-pattern|]]]
```

-body->

The following theorem says that if something of the form [alive x] is erased then [dead x] should be asserted.

```
<erasing [x]
        [alive _x]
        <assert [dead .x]>>
```

5.2 PLANNER Functions

5.2.1 Data Primitives

Some of the functions in PLANNER are given below together with brief explanations of their purpose Examples of their use are be given immediately after the definition of the primitives below. The primitives probably cannot be understood without trying to understand the examples since the language is highly recursive. In general PLANNER remembers everything that it is doing on all levels unless commanded to forget some part of this information. The default response of the language when a simple failure occurs is to backtrack to the last decision that it made and to make another choice.

<CANDIDATES

|kind| |pattern| |state-path|> are the |kind| candidates that have the same coordinates as |pattern| and are in the local data base defined by |state-path|. CANDIDATES is the basic retrieval function for the data base. The candidates can be generated incrementally if it is not desired to construct them all at once at the beginning. The kind of data retrieved may be:

```
CURRENT for assertions
FUNCTION for functions
```

5.2.1.1 Assertions

<ASSERT!-TENTATIVE
|statement| |rec| [¬"PATH |state-path|] [¬"ALREADY" |already-current|]> puts |statement| in the data base defined by |state-path| and tries to draw conclusions according to the recommendation |rec|. Recommendations are optional; the default recommendation is [¬"TRY"] which says not to try any theorems. If the statement is already in the data base then |already-current| is evaluated. If the value of |already-current| is ¬"REASSERT" then the |statement| is asserted in the first element of |state-path|. The ¬"reassert" feature is due to Drew McDermott. Otherwise, the function ASSERT causes the statement statement with properties to be inserted in the data base which is the first element of |state-path|. Then

```
<CALL
        [<[ASSERT |statement| ]>
                |state-path|
                |rec| ]>
```

is evaluated to draw conclusions from statement. If the call to DRAW ultimately fails then |statement| is removed from the data base. The argument |already-current| is due to Peter Bishop. The recommendation is optional. The value of the function ASSERT is the arc from the state which contains the assertion having as indicator the assertion.

```
<assert
        <put
                [subset a b]
                [difficulty trivial]>>
```

asserts that the set a is a subset of the set b and put the value trivial under the indicator difficulty.

<ASSERT!-PERSISTENT

|statement| |rec| [-"PATH |state-path|] [¬"ALREADY" |already-current|]> is exactly like ASSERT!-TENTATIVE except that |statement| is not withdrawn from |state-path| on backtracking.

Expressions of the form <CLAUSE [declarations] -alternatives-> denotes an assertion with variables declared followed by logical alternatives. For example

```
                    <assert
                           <clause [[<set> x y z]]
                                   [not [subset ?x ?y]]
                                   [not [subset ?y ?z]]
                                   [subset ?x ?z]>>
```

asserts in declarative form that the subset relation is transitive for sets. In other words it is equivalent to

```
                    <assert
                        <clause [[<set> x y z]]
                            [implies
                                [and
                                    [subset ?x ?y]
                                    [subset ?y ?z]]
                                [subset ?x ?z]]>>
```

Another kind of assertion is one which has variables which are consumed by being bound For example if we translate the assertion that John is somewhere as <assert <closure <clause [ ] [at John ?x]> x>>, then <goal [at John store]> causes x to be bound to the atom store. Thereafter <goal [at John home]> fails since the identifier x was consumned in being bound to the atom store. The above problem was suggested by Gene Charniak.

W. Bledsoe suggested trying the problem of showing that [all a [some b [p b a]]] follows from [some x [all y [p x y]]].

```
                <assert <clause [y] [p [x0] ?y]>>
                <prog [b]
                        <goal <clause [p ?b [a0]]>>>
                        b gets the value [x0]
```

The expression <clause [y] [p [x0] ?y]> is the assertion Skolem form of the assertion [some x [all y [p x y]]] where x0 is the Skolem function for x. The expressions <clause [p ?b [a0]]> is the goal Skolem form of [all a [some b [p b a]]] where a0 is the Skolem function for a. On the other hand if we were to try to derive [some x [all y [p x y]]] from [all a [some b [p b a]]] we would fail:

```
                <assert <clause [a] [p [b0 ?a] ?a]>>
                <prog [x]
                        <goal <clause [p ?x [y0 ?x]]>>>
```

The identifier x cannot be be bound. The many-sorted omega order quantificational calculus of PLANNER allows for the possibility of null domains. For example it does not follow that there is a god which is a deity if we assume that all gods are deities. That is [some [in g god] [deity g]] does not follow from [all [in g god] [deity g]]. Thus we cannot prove the existence of a god so easily. However [some [in g god] [deity g]] does follow from [some [in g god] [mythical g]] and [all [in g god] [implies [mythical g] [deity g]]].

```
        <assert [mythical [g0]]>
        <assert
            <clause [[<god> g]]
                [not [mythical ?g]]
        [deity ?g]>>
<prog [[<god> x]]
        <temprog [ ]
                <assert <clause [ ] [not [deity ?x]]>>
                ;"assert that there are no gods which
                        have the property of being deities"
                <prog [literal1 literal2]
                        <current
                            <clause
                                <all
                                        [deity <?>]
                                        _literal2>
                                <?>>>
                        <current
                            <clause
                                <all
                                        [not [deity <?>]]
                                        _literal1>
                                <box>>>
                        <assert
                            <resolve
                                .literal1
                                .literal2>>>
                ;"resolve a clause which contains an element
                        which matches [deity <?>] and
                        a singleton clause
                        whose element matches
                        [not [deity <?>]] producing
                        <clause [ ] [not [mythical ?x]]> which
                        is then asserted"
                <prog [literal1 literal2]
                        <current
                            <clause
                                <all
                                        [mythical <?>]
                                        _literal1>
                                <box>>>
                        <current
                            <clause
                                <all
                                        [not [mythical <?>]]
                                        _literal2>
                                <box>>>
                        <assert
                           <resolve
                                .literal1
                                .literal2>>>
                ;"resolve two singleton clauses;
                        one containing
                        a positive instance of mythical and
```

```
                              one a negative instance.
                              this binds x to [g0] and produce a

                              clause which is written <box>"
                        <current <box>>
                        ;"thus we have derived the null clause which
                              is a contradiction">
                  <assert <clause [] [deity .x]>>>
                  ;"assert [deity [g0]]"
```

5.2.1.2 Erasures

<ERASE!-TENTATIVE

|statement| |rec| [¬"PATH" |state-path|] [¬"NOT-FOUND" |not-found|]> tries to find an assertion |a| in |state-path| in the data base that matches |statement|. If such an assertion |a| is found then it is erased and

```
                  <CALL
                        [<[ERASE |a|]>
                              |recommendation|
                              |state-path|]>
```

is evaluated to assay the implications of the change. If no such assertion is found then |not-found| is evaluated. If the change statement fails or if a failure backtracks to the function ERASE, then |a| is reinserted in the data base and the whole process repeats with another statement from the data base. The value of the function ERASE is an arc from an element of |state-path| with indicator a statement which matches |pattern|. The reader should be careful not to confuse what happens when the function ERASE is called to remove something from the data base with what happens when an ASSERTION fails and thus removes what was asserted from the data base. The function ERASE may attempt to do pattern directed invocation to deduce consequences of the deletion whereas ASSERT will not. The argument |not-found| is due to Peter Bishop.

<erase [on-top-of brick1 brick2]> erases the fact that brick1 is on top of brick2.

<ERASE!-PERSISTENT

|statement| |rec| [¬"PATH" |state-path|] [¬"NOT-FOUND" |not-found|]> is exactly like the function ERASE!-TENTATIVE except that the assertion deleted from |state-path| is not re-inserted on backtracking.

5.2.1.3 Goals

<CURRENT?

|pattern| |state-path|> tests to see if a statement that matches |pattern| currently is in |state-path|. If there is such a statement, then the identifiers in |pattern| are bound to the appropriate values. If there is no such statement, then CURRENT? returns false. If a simple failure backtracks to the function CURRENT, then the identifiers that were bound are unbound, Then the whole process repeats with another statement in the data base.

PLANNER is designed so that the time that it takes to determine whether a statement that matches pattern is in the data base or not is essentially independent of the number of irrelevant statements that have already been asserted. A coordinate of a structure is defined by some atom, number, or string being in some position of the structure. When an s-expression is asserted PLANNER remembers every coordinate that occurs in the s-expression. Two expressions are similar on retrieval only to the extent that they have the same coordinates. The function <MERGE |w| |l|> will merge |w| into the list |l|. Consider the simple assertion

<assert .z [¬"path" (.s1)]> where s1 is bound to a state and z is bound to ¬[a [b c]] causes the following changes:

```
<put
        <position 1 current>
            [a
                <merge
                         .z
                         <get a
                                  <position 1 current>
                                  (0)
                                  ;"if the bucket is empty then,
                                           initialize it with
                                          an empty list">> ]>

<put
        <position 1 <position 2 current>>
            [b
                <merge
                         .z
                         <get b
                                  <position
                                       1
                                      <position 2 current>>
                                  (0)>> ]>

<put
        <position 2 <position 2 current>>
            [c
                <merge
                         .z
                         <get c
                                  <position
```

```
                                        2
                                       <position 2 current>>
                                    (0)>>]>

        <put .s1 [.z ¬"assert ed"]>
```

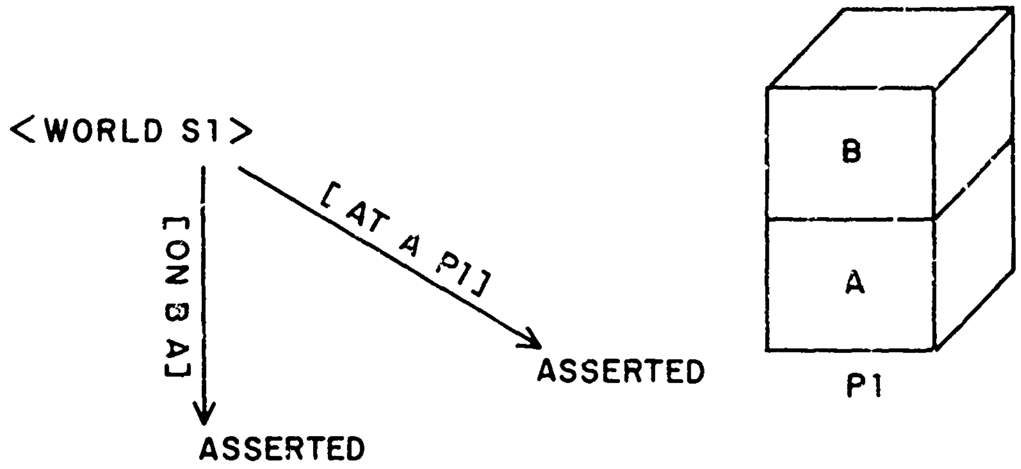
Classes are stored in buckets under the position ¬"class". Thus the assertion <assert .w [¬"path" (.s1)]> where w is bound to [nonempty <class e f>] would result in:

```
        <put
                <position 1 current>
                    [nonempty
                        <merge
                                .w
                                <get nonempty
                                        <position 1 current>
                                        (0)
                                        ;"if the bucket is empty then,
                                                initialize it with
                                               an empty list">>]>

        <put
                <position ¬"class" <position 2 current>>
                    [e
                        <merge
                                .w
                                <get e
                                        <position
                                            ¬"class"
                                           <position 2 current>>
                                        (0)>>]>

        <put
                <position ¬"class" <position 2 current>>
                    [f
                        <merge
                                .w
                                <get f
                                        <position
                                            ¬"class"
                                           <position 2 current>>
                                        (0)>>]>

        <put .s1 [.w ¬"asserted"]>
```

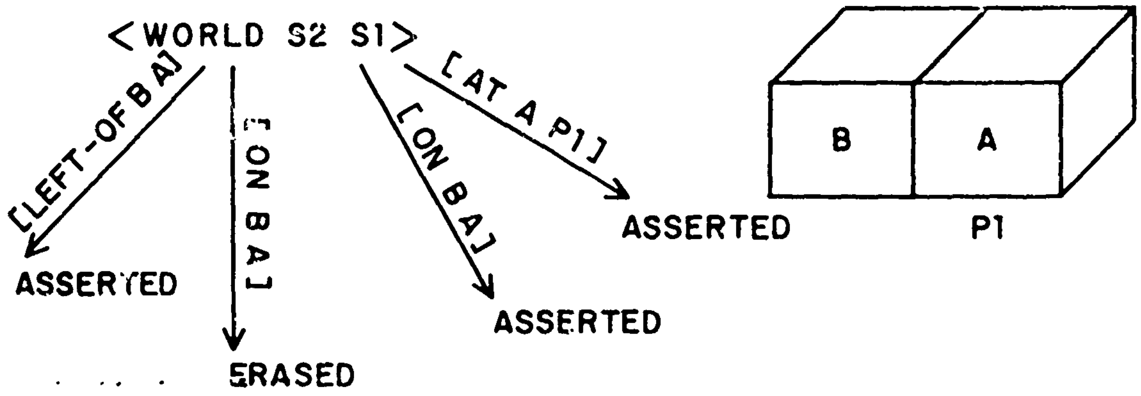
Clauses are classes at their top level. For example the clause

5.2 page 221a

TREE-STRUCTURED WORLDS



INITIAL WORLD WITH B ON A WHICH IS AT POSITION P1



PUSHED DOWN WORLD WHERE B HAS BEEN MOVED TO THE LEFT OF A. NOTE THAT A IS STILL AT P1 FROM THE POINT OF VIEW OF THIS WORLD. HOWEVER B IS NO LONGER ON A.

<clause [] [not [on a b]] [on a c]> would be stored under the coordinates for [not [on a b]] and [on a c]. Variables in expressions are ignored on indexing. Thus two expressions which are the same except for change of variables are considered equivalent. When the bucket under some coordinate exceeds a threshold then the bucket could be sub-divided by taking the coordinates by pairs. The only reason that we don't store statements under all the possible combinations of coordinates is that we can not afford to use that much space. Storing the most recent assertion at the front of a bucket also tends to speed retrieval. If a total ordering is imposed on the assertions, then the buckets can be sorted. Richard Greenblatt has constructed a clever total ordering on the assertions which also has the advantage of storing new assertions at the front of the buckets. The total ordering is constructed incrementally as assertions are made. If MATCHLESS had an efficient parallel processing capability then the retrieval could be even faster since we would do the look-ups on coordinates in parallel. We might imagine a machine with multiple program counters each of which is capable of interrupting the execution of the others. However, with the current technology it appears more economical to timeshare a few very fast physical processors. Clauses are stored in a special way for efficiency. The value of the expression <CURRENT |pattern| |state-path|> is an arc from the state in |state-path| which contains the assertion with indicator name being an assertion that matches |pattern|.

```
                <current?
                        [subset a b]
                        [¬"use" <has [difficulty trivial]>]>
```

is true only if it has been proved that a is a subset of b with the value trivial under the indicator difficulty. We shall use the prefix operator ?x for <GIVEN x> to denote variables of the quantificational calculus. The concept of a variable is different from that of an identifier in that variables have global scope.

given:

```
                <assert
                        <put
                                <clause [[<object> x] [<set>  y z]]
                                        [subset [f ?x] ?y]
                                        [subset ?y ?z]>
                                [difficulty hard]>>
```

The above statement says that for all objects x and sets y z that [f x] is a subset of y or y is a subset of z. evaluate:

```
                <prog [[<set> w u] ]
                        <current
                                <clause [subset _w _u] <?>>>>
evaluates
                        to <clause
                                [[<object> x]]
                                        [subset [f ?x] [f ?x]]>
                w gets the value [f ?x]
                u gets the value [f ?x]
```

<CURRENT

|pattern| |state-path|> is exactly like CURRENT? except that if it runs out of objects that are currently in |state-path| which match |pattern| then it generates a simple failure instead of returning false. The value of CURRENT is the node which is the

property list of an assertion in |state-path| which matches |pattern|

```
        <GOAL
                |goal-pattern| |rec| [¬"PATH" |state-path|]]> tries to
```

achieve the |goal-pattern| according to a recommendation |rec|. Recommendations are optional; the default recommendation is [¬"USE" ¬"CURRENT" <?>] which means the data base is searched to see if there is something already proved which matches |goal-pattern| then use it otherwise try any consequent theorem whose consequent matches |goal-pattern|. The recommendation |rec| must be of one of the following two forms:

```
1: [¬"USE"
        ¬"CURRENT"
        -pats-] is equivalent to

<COND
    [<CURRENT? |goal-pattern| |state-path|>]
    [¬"ELSE"
        <CALL
                [<[GOAL |goal-pattern|]>
                        [¬"USE" -pats-]
                        |state-path|]>]>

2: [¬"USE1"
        ¬"CURRENT"
        -pats-] is equivalent to

<COND
    [<CURRENT? |goal-pattern| |state-path|>]
    [¬"ELSE"
        <CALL
                [<[GOAL |goal-pattern|]>
                        [¬"USE" -pats-]
                        |state-path|]>]>
```

The ¬"USE1" recommendation is due to Pat Winston. Alan Kay has suggested that the syntax of PLANNER could be easily changed so that every expression is a goal. Thus instead of writing <GOAL x> we would simply write x. Alan's suggestion has the merit that it simplifies the language. One reason that we do not do this is that pattern directed invocations are somewhat more inefficient than straightforward calls in which the name of the called function is explicit. Anyone who prefers the other syntax can easily expand all function calls <f args> into <[f args]> by a trivial macro.

Suppose that we know that zero is an integer and that if n is an integer then n+1 is an integer. We would like to find an integer j which is not zero.

```
<assert [integer 0]>

<assert <consequent [n]
        [integer [+ ?n 1]]
        <goal [integer ?n]>>>

<prog [[<non 0> j]]
        <goal [integer ?j]>>

        j gets the value [+ 0 1]
```

<GOAL?

|goal-pattern| |rec| [-"PATH" |state-path|]|> is exactly lieke GOAL except that it returns <> instead of backtracking if it runs out of alternatives.

<GOALS>

returns as its value a list of the specifications of the currently active goals.

<SUBGOAL

-clauses-> attempts to match the first element of each clause in turn to the elements of the list of currently active goals.

If the first element of a clause matches then execution continues with the remaining elements of that clause.

5.2.2 Control Primitives

<SWITCH
|new-state-path| |expression|> evaluates |expression| using the |new-state-path| to do retrievals from the data base. At any given time PLANNER expressions are being evaluated in a state path. A top level process begins by using the primary data base as its state. It can switch into a local state by using the the function SWITCH. Tree structures of local states can be created by using the function STATEPROG. States can be conceptualized as a linear list of changes to the data base. Thus there can be several incompatible states of the world simultaneously under consideration. Although the tree structure of the local states can be conceptualized as a linear list of changes, it is actually implemented more efficiently so that the retrieval time for assertions is essentially independent of the size and number of local states. The assertions in the data base are tagged as to which states they are in.

<STATE>
returns as its value a new local state.

<PRIMARY>
is the primary state of the system.

<UPDATE

|state1| |state2|> updates |state1| into |state2|. If the second argument is missing the global data base is assumed.

<GATE

|x|> is the value of |x| unless |x| fails simply in which case it is <>. The expression &|x| is an abbreviation for <GATE |x|>.

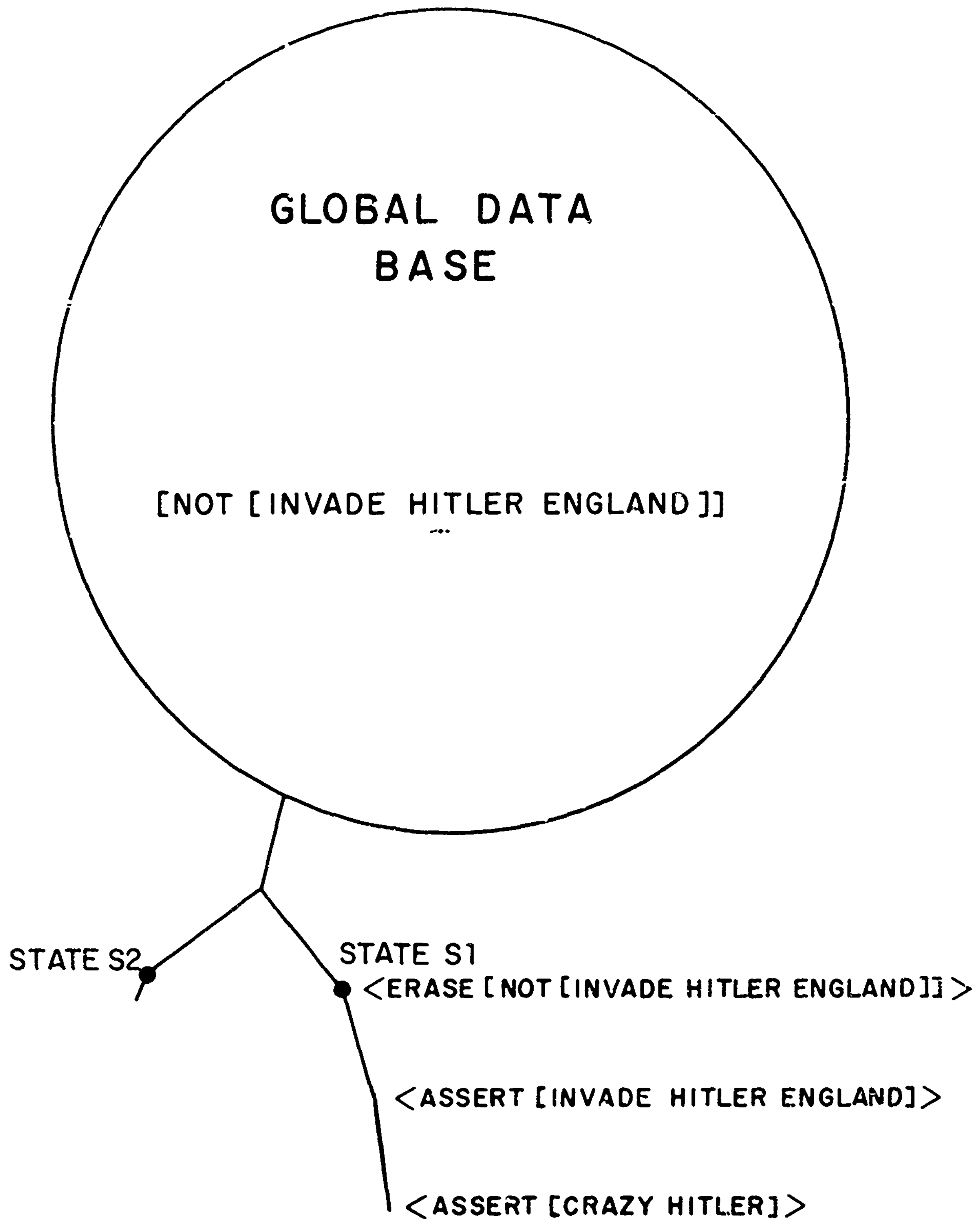
```
!%<block (<oblist gate!-> <oblist>)>
<define gate <function out ['x]
        <failpoint [ ] <>
           [message activation?]
                <cond
                    [<not <or .message .activation?>>
                        ;"neither the message nor
                                activation are on"
                        <.out <>>
                        ;"exit gate with false"]>>
        <eval .x>
        ;"the value of gate is the value of .x unless
                the evaluation of x fails>>
!%<end-block>
```

```
<cond [¬"else" <fail>]> fails with the message <>.
<cond [<fail> 3] [¬"else" 7]> fails
<cond [&<fail> 3] [¬"else" 7]> evaluates to 7.
<cond [<> 3]> evaluates to <>.
<cond [<> 3] [¬"else" 4]> evaluates to 4.
<cond [¬"else" <fail>] [¬"else" 5]> fails.
<cond [&¬"else" <fail>] [¬"else" 5]>  evaluates to 5.
```

<TEMPROG

+checker+ +activation-name+ [-declarations-] -body-> is like the function PROG except all assertions and erasures that are made within the scope of the function TEMPROG are undone when the function TEMPROG returns. The function TEMPROG is useful for dealing with hypotheticals. Suppose that we wanted to establish [all x [p x]]

"HITLER WOULD HAVE BEEN CRAZY TO INVADE ENGLAND"


GLOBAL DATA BASE
[NOT [INVADE HITLER ENGLAND]]
STATE S2
STATE S1
<ERASE [NOT [INVADE HITLER ENGLAND]]>
<ASSERT [INVADE HITLER ENGLAND]>
<ASSERT [CRAZY HITLER]>

b mathematical induction.

```
                <goal [p 0]>
                ;"first try to prove [p 0]"
                <temprog [k <arbitrary <integer>>]
                        ;"let k be an arbitrary integer"
                        <assert [p .k]>
                        ;"assert that p holds for k"
                        <goal [p !'<+ .k 1>]>
                        ;"try to prove that p holds for k+1">
```

<SWITCH

|state-path| |expression|> causes |expression| to be evaluated with |state-path| as its current local state path. The value of PATH!-STATE is the current state path. Local states are useful for handling contra-to-factual conditionals and for simultaneously manipulating inconsistent states of the world. Assertions affect only the state which is the first element of the state path in which the assertion is evaluated. The following assigns the identfier s1 the value which is a local state path in which Hitler invaded England.

```
        <switch
                <_ :s1 [<state> !.path!-state]>
                <assert [invade Hitler England]>>
```

We further suppose that Hitler is crazy. This could be expressed by doing the assertion within s1 and assigning the result to s2:

```
        <switch
                <_ :s2 [<state> !.s1]>
                <assert [crazy Hitler]>>
```

Now if we ask if Hitler is crazy in the state path s1, the answer is that he is not; but he is crazy in the state path s2.

```
        <switch .s1 <current [crazy Hitler]>> fails
        <switch .s2 <current [crazy Hitler]>> is true
        <switch .s2 <current [invade Hitler England]>> is true
        <switch
                [<1 .s2>]
                <current [invade Hitler England]>> fails
        <switch
                [<1 .s2>]
                <current [crazy Hitler]>> is true
        <switch
                [<2 .s2> <1 .s2>]
                <current [crazy Hitler]>> is true
```

Erasures affect the first local state of the state path in which they are evaluated. After

```
        <switch .s1 <erase [invade Hitler England]>> we have


        <switch .s1 <current [invade Hitler England]>> fails
        <switch .s2 <current [invade Hitler England]>> fails
```

If we know that a formula of the form [or |x| |y|] is true and we want to establish a goal of the form |g| then we could write:

```
<PROG [ ]
        <TEMPROG [ ]
                <ASSERT |x|>
                <GOAL |g|>>
        <TEMPROG [ ]
                <ASSERT y>
                <GOAL |g|>>
        <ASSERT |g|>>
```

The above form of disjunction elimination is often used when y is of the form [NOT |x|]. Goals of the form [or |x| |y|] can be established as follows:

```
<PROG [ ]
        <TEMPROG [ ]
                <ASSERT [NOT |x| ]>
                <GOAL y>>
        <ASSERT <CLAUSE [ ] |x| |y|>>>
```

5.2.2.1 Failure Primitives

<UNIQUE> fails if the current goal is not unique among all the goals that are currently active.

<UNIQUE

<|p| -args-> |place|> fails if the procedure |p| with arguments -args- is not unique among all the procedures that are active in |place|. The |place| can be a process or it can be [BETWEEN |name1| |name2|] in which case only the procedures between |name1| and |name2| are be examined.

<RETRY

|activation|> causes failure to |activation| which must include the call to RETRY within its scope. Execution resumes with the beginning of the named block.

```
<prog here [a]
        <_ _a 3>
        <prog there [ ]
                <cond
                    [<is? 4 .a>
                        <.here .a>
                        ;"exit .here with .a"]>
                <_ :a 4>
                <retry .there>>> evaluates to 4
```

5.2.2.2 Finalize primitives

```
<FINALIZE
            +activation-name+>
```
causes all actions that have been taken in the block +activation-name+ to be finalized and then returns the value of +activation-name+. Thus <<FINALLIZE +activation-name+> -values-> will finalize all the actions that have been taken in the scope of +activation-name+ and then exit +activation-name+ with -values-. Actions which are finalized are not undone if a failure backs up. Finalization can be used to save storage for actions which should not be automatically reverted in case of failure. For example, robot thinking for a given task is often divided into two phases: a planning phase and an action phase. In PLANNER this is typically done by having the planning phase return as its value a PROCEDURE which is to be executed in the action phase. Asser ions which record events which have taken place in the "real world" should be finalized in the action phase as they happen.

5.2.2.3 Repetition Primitives

```
<FOR
        +checker+ +activation-name+ [-declarations-]
        [-for-specifications-
        [¬"CURRENT" |pattern| |state-path|]]
        -body->
```
is the for statement of PLANNER. For each assertion in the data base that matches |pattern| the -body- is executed. For example the following statement places all the bricks on brick1 in the blue box.

```
        <for
                        [[<brick> x]]
```

```
                    [[¬"current" [on-top-of _x brick1]]]
                <pick-up .x>
                <place-in <' [blue box]>>>
```

```
<PERSIST
        +checker+ +activation-name+ [-declarations-]
        [[¬"INITIAL" -initial-action-]
        [¬"TEST" |test| -test-action-]
        [¬"LIST" |item| condition]
        [¬"STEP" -step-action-]
        [¬"FINAL" -final-]]
        -body->
```

where +activation-name+ and +checker+ are optional is equivalent to

the following:

```
        <PROG +checker+ +activation-name+
                [-declarations-
                [COLLECTED ()]]
                ;"initialize COLLECTED to []
                <FAILPOINT
                    [MESSAGE ACTIVATION]
                        <COND
                            [<NOT? <OR?
                                        .MESSAGE
                                        .ACTIVATION>>
                                -final-
                                <.+activation-name+
                                        .COLLECTED>
                                ;"exit .+activation-name+ with
                                .collected"]>>
                -body-.
                <COND
                    [|test|
                        -test-action-
                        <.+activation-name+  .COLLECTED>
                        ;"exit .+activation-name+
                                with .collected"]>
                <COND
                    [|condition|
                        ;"if the condition is met
                                then add item
                                to the end of COLLECTED"
                    <_ :COLLECTED (!.COLLECTED item)>]>
                -step-action-
                <FAIL>
                ;"generate a simple failure">
```

"Are all the blocks in box1 green?" translates to

```
            <persist b1 [[<block> b]]
                [[¬"final" <.b1 t> ;"exit .b1 with t"]]
                <goal [in _b box1]>
                ;"find a block in box1"
                <cond
                    [&<goal [green .b]>
                        ;"if the block is green then
                                continue with the loop"]
                    [¬"else"
                        <fail <> .b1>
                        ;"otherwise generate a failure out of
                                the persist loop"]>>
```

```
        <FIND
            +activation-name+
            [-declarations-]
            [
                [¬"QUANTITY |quantity|]
                [¬"LESS" |lower-bound| -fewer-]
                [¬"GREATER" |upper-bound| -more-|]]
            |item|
            -body->
```

constructs a list of between |lower| and |upper| |item|s according to the |body|. The FIND function is equivalent to the following:

```
            <STRAIGHTEN <PROG +activation-name+
                [-declarations- [NUMBER 0] [COLLECTED ()]]
                <failpoint [] <> [M A]
                    <COND
                        [<NOT? <OR? .M .A>>
```

```
                                                                <COND
                                                                    [<NOT? <IS?
                                                                                 |quantity|
                                                                                 ¬"ALL">>
                                                                         <FAIL>]>
                                                                ;"if the quantity sought
                                                                         is not all
                                                                         then backtrack"
                                                                <COND
                                                                    [<IS?
                                                                         <LESS |lower-bound|>
                                                                         .NUMBER>
                                                                         -less-]>
                                                                <.+activation-name+
                                                                         .COLLECTED>
                                                                ;"return with the items
collected"]>>
                                             -body-
                                             <_ :COLLECTED (!.COLLECTED |item|)>
                                             <INC!-PERSISTENT NUMBER>
                                             <COND
                                                 [<IS? |quantity| .NUMBER>
                                                      <.+activation-name+ .COLLECTED>
                                                      ;"if have found the quantity
                                                                desired then return them"]>
                                             <COND
                                                 [<IS? <GREATER |upper-bound|> .NUMBER>
                                                      -more-]>
                                             <FAIL>>>
```

"Find three boxes that contain green blocks."

translates to:

```
<find [[<box> x] [<block> b]] [[¬"QUANTITY" 3]] .x
          <goal [box _x]>
          <goal [contains .x _b]>
          <goal [green .b]>>
```

5.2.2.4 Multi-Process Primitives

In more complicated situations, we find that it is convenient to be able to have more than one PLANNER process.

```
<FAIL
```

|message| |place| |function|> generates a failure with |message| to the |place| at the last point that execution left |place|. If the process which called FAIL is ever resumed with arguments, then it begins by applying |function| to the arguments.

```
<EXHAUST
        +checker+ +activation-name+ [-declarations-]
        [[¬"INITIAL" -initial-action-]
        [¬"TEST" |test| -test-action-]
        [¬"ACTION" -action-]
        [¬"LIST" |item| |condition|]
        [¬"STEP" -step-action-]
        [¬"FINAL" |final|]]
        -body->
```

attempts to execute -body- once for each time that -action- is successfully evaluated. Every time that the body it executed the function EXHAUST sends a simple failure to the action to see if it has any alternatives. An EXHAUST loop is very much like a PERSIST loop which is defined above. Both loops are driven by the failure mechanism. The main difference is that the effects of executing the body [illegible] PERSIST loop are not preserved because a failure must propagate through the body before it can be executed again. In an EXHAUST loo[illegible] separate process is created for the action so that the effects of [illegible] the body can be preserved. The function EXHAUST is equivalent to the following expression:

```
<PROG +checker+ +activation-name+
    [
        [COLLECTED ()]
        [<proc>
                [ACTION-PROCESS <PROCESS ,ACTION-FUNCTION>]
                [VAL-PROC <.ACTION-PROCESS <PROCESS>>]]]
        ;"declare COLLECTED to be initialized to [ ]"
        ;"ACTION-PROCESS is the name of the
                process which is to be exhausted by failure"
        ;"start the PLANNER process .ACTION-PROCESS in
                which the action is executed with
                the name of this process
                as an argument so that it can later resume
                this process"
        ;"we expect one value to be returned
                which we shall call VAL-PROC"
        <REPEAT [ ]
                <COND
                    [<IS? EXHAUSTED .VAL-PROC>
                        -final-
                        <.b .COLLECTED>
                        ;"exit .b with .collected"]
                    [|test'
                        -test-action-
                        ;"if the test is met
                                then execute the test-action"
                        <.b .COLLECTED>]>
                -body-
                <COND
                    [|condition|
                        <_ :COLLECTED (!.COLLECTED |item|)>]>
                ;"if the condition is met then add the item to the end
                        of the list of collected items"
                <FAIL
                        <>
                        .ACTION-PROCESS
                        <FUNCTION [Y] <_ :VAL-PROC .Y>>>
                ;"suspend
                        execution of the current process
                        and begin failing from the point within
                        the action process
                        where execution last left off">>
```

The following function is defined so that we can start off the evaluation of the action process.

```
<DEFINE ACTION-FUNCTION
    [FUNCTION [[<proc> MAIN]]
        <FAILING? [<?> <.MAIN EXHAUSTED>]>
        ;"when the action finally is exhauseted
                resume the process .MAIN with the value EXHAUSTED and
                terminate the action process"
        -action-
        <.MAIN SUCCESS>
        ;"resume the main process with the value SUCCESS"]>
```

Suppose that we want to disprove a proposition .p using likely counterexamples. Furthermore we would like to work on each counterexample in parallel as it is found.

```
<exhaust disprove [c]
        <goal [likely-counter-example _c .p]>
               ¬|<cond
                   [&<goal .c>
                       <temporize
                               .disprove>
                               ¬"found-counter-example">>]>>
```

5.3 Clauses in PLANNER

We would like to explore the potentialities for using PLANNER to control a resolution based deductive system. Since the question whether or not a given formula is a theorem or not is undecidable, a complete proof procedure using resolution for the first order quantificational calculus must in general be rather inefficient. In fact any uniform proof procedure for the first order quantificational calculus can be sped up by an arbitrary recursive function for almost all proofs. The result on the necessary inefficiency of a complete proof procedure should be sharpened up. New theoretical tools must be developed in order to make any substantial advance on the problem. The importance of resolution as a problem solving technique does not lie in the fact that it appears to be the fastest known uniform proof procedure for first order logic. Rather, resolution provides one technique for dealing with the logic of disjunction and instantiation. Domain dependent procedures must provide most of the direction in the computation to attempt to prove a theorem. We shall introduce new actors to match clauses:

<CLAUSE

-patterns- |rest-of-disjuncts|> matches a clause only if it has disjuncts which match -patterns- and the rest of the disjuncts match the pattern |rest-of-disjuncts|.

```
<prog [y]
      <_
                  <clause [subset a :y]>
                  <clause [x] [subset ?x b]>>>
y gets the value b
x gets the value a
```

<CLAUSE-OF

|pat| |disjuncts-that-match-pat| |rest-of-disjuncts|>

matches a clause such that the clause of the disjurcts that match |pat| in turn match the pattern |disjuncts-that-match-pat| andthe clause of the the rest of the disjuncts match |rest-of-disjuncts|.

The following functions are used to manipulate clauses.

<CLAUSE

[-declarations-] -disjuncts-> returns a copy of a clause with the variables declared.

<VARIABLES

|clause|> returns the variables in the clause.

<INSTANTIATE

|clause|> returns a copy of the clause with all of its variables instantiated with unique constants of the appropriate type.

<RESOLVE

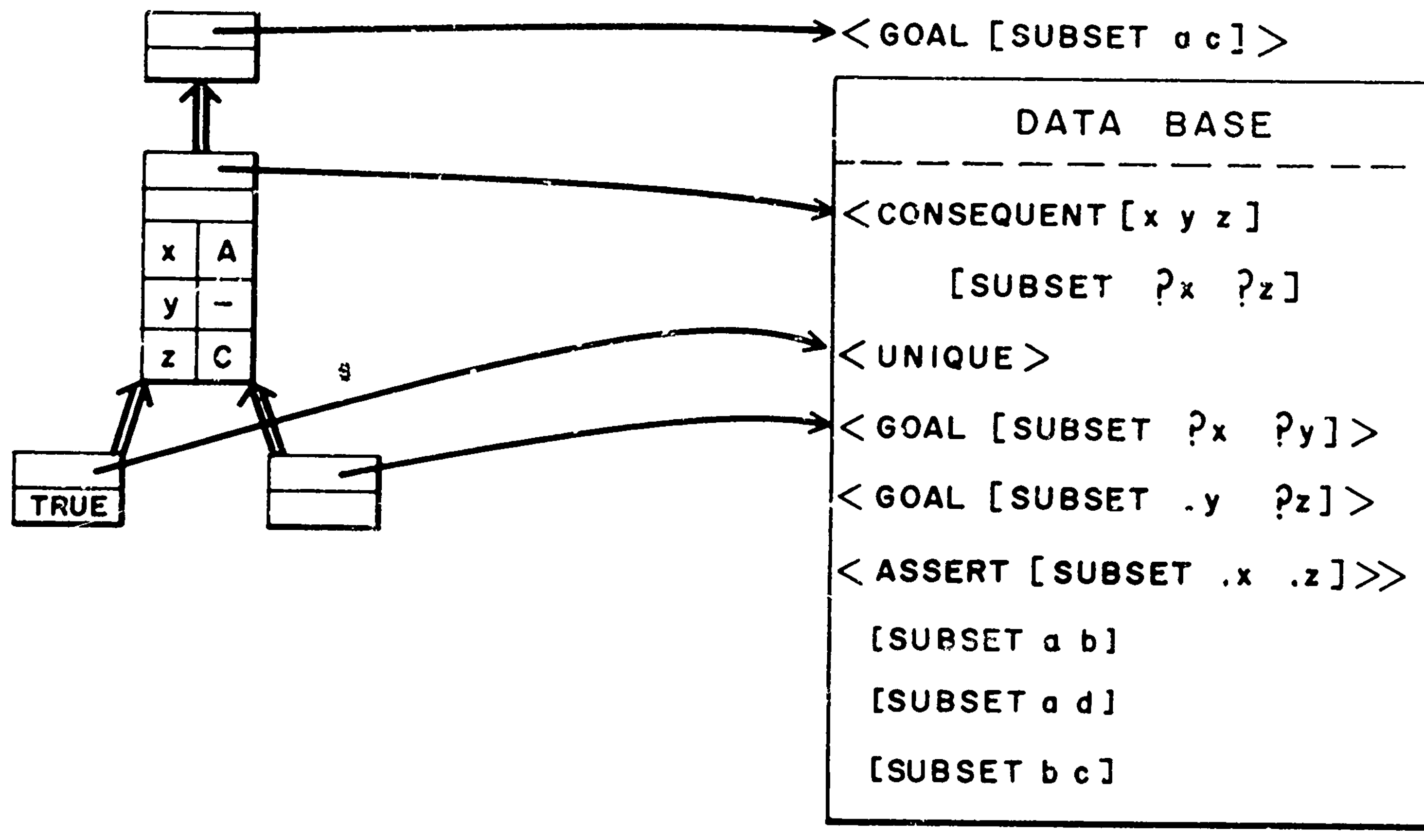
-clause-specifications-> results in resolving the clauses represented by the clause specifications together to yield a clause which is returned as the value of the function resolve. A clause specification is the literal of the clause which is to be unified.

```
<FOR-RESOLVENT
        +checker+ +activation-name+ [-declaratiions-]
        [[¬"CLAUSES" -clause-specifications-]
        [¬"RESOLVENT" |resolvent| ]
        -for-loop-specifications-]
        -body->
```

attempts to execute the body of the for statement once for each result of resolving clauses that meet the clause specifications to produce a clause which matches the pattern resolvent.

It is possible for PLANNER to run out of things to evaluate before it has deduced the null clause. A complete proof procedure could be called to try to finish off the proof. If in the course of its operation, the complete procedure generates a clause that matches the antecedent of a theorem then PLANNER can be re-invoked. The complete procedure could be run in parallel with PLANNER. Thus using PLANNER we could implement a complete proof procedure. The point is that implementing any "reasonable" proof procedure should be easy in PLANNER. However, we should not rely on a uniform proof procedure to solve our problems for us.

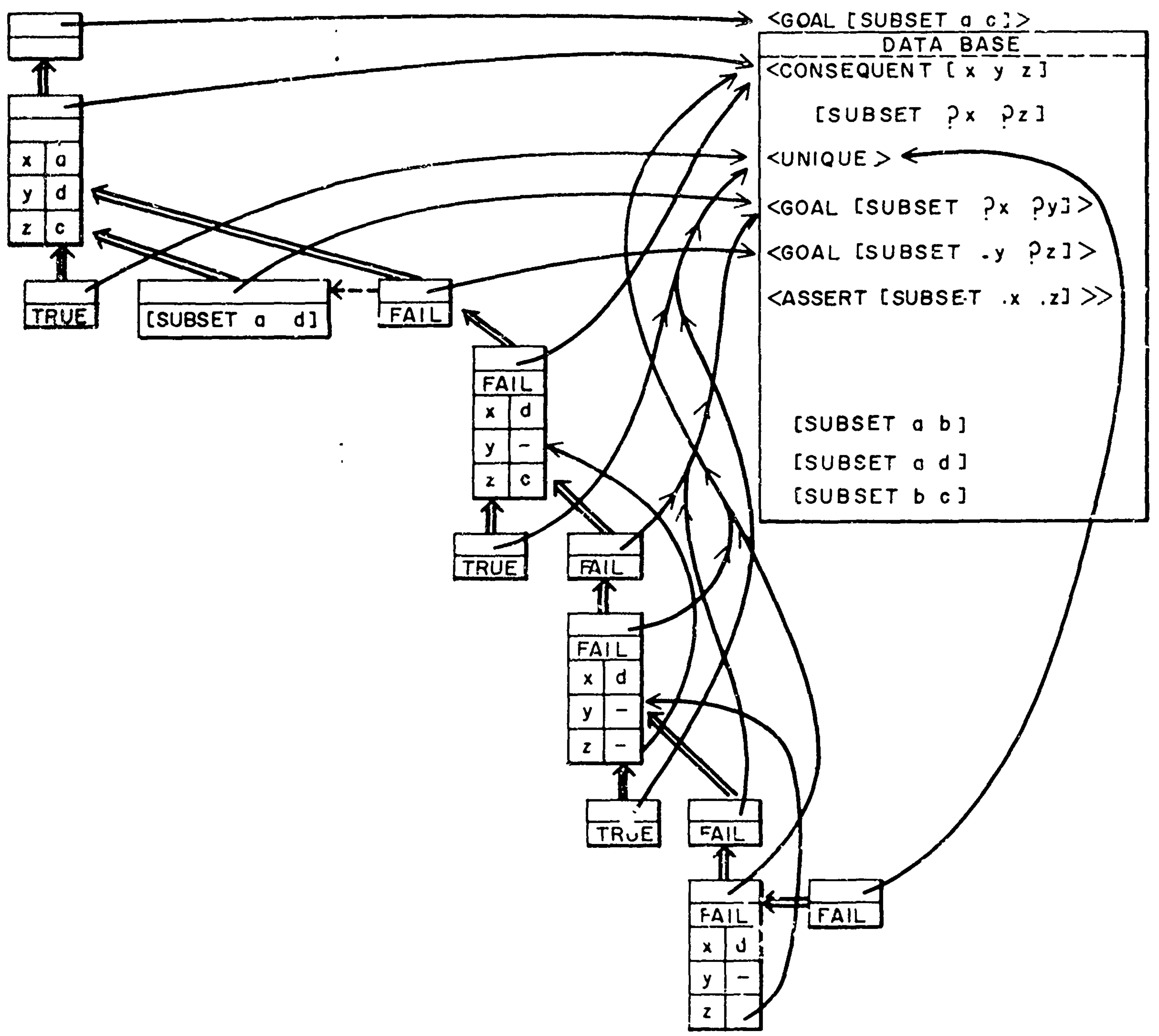
SNAPSHOT NO. 1



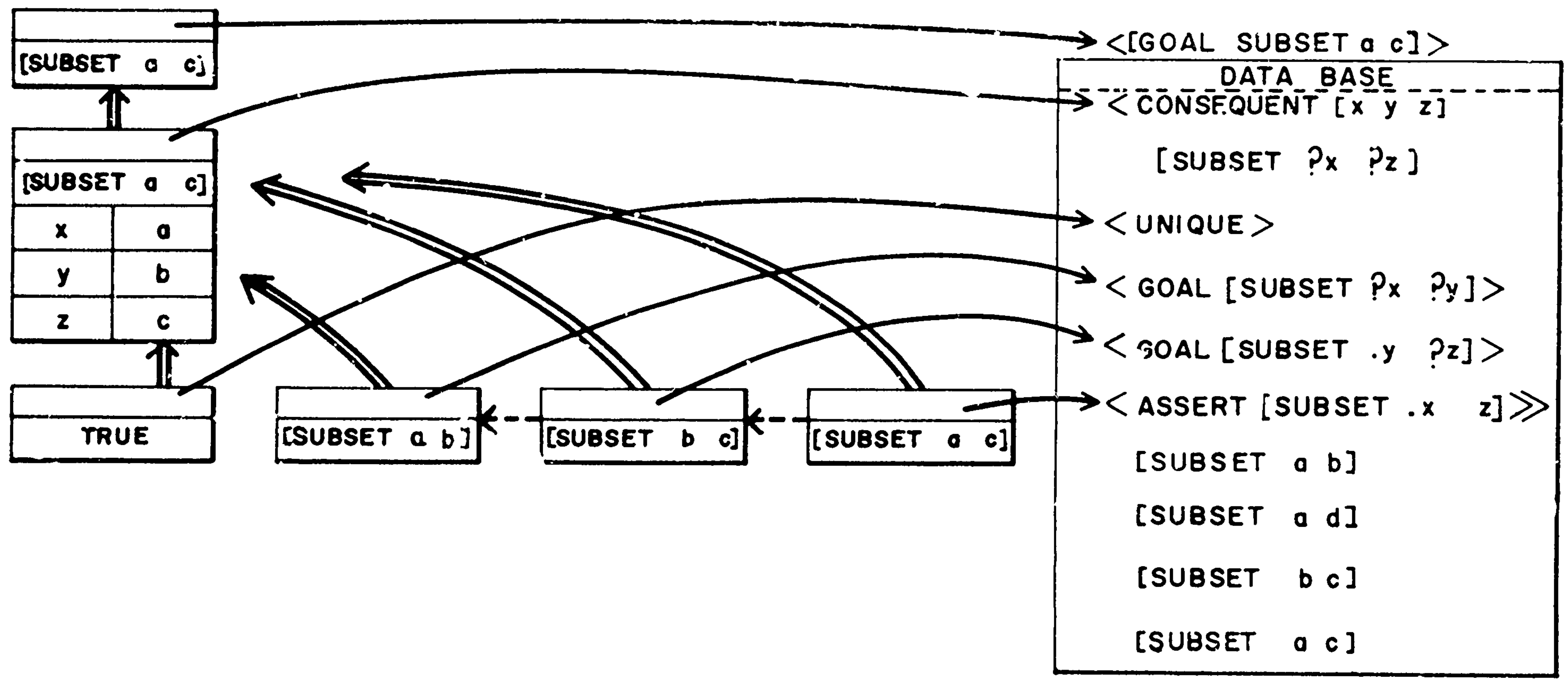
53 page 240a

5.3 page 240b

SNAPSHOT NO. 2



SNAPSHOT NO. 3



5.3 page 240a

5.4 A Simple Example

5.4.1 Using a Consequent Theorem

Suppose that we know that [subset a b], [subset a d], [subset b c], and [all [function <boole> [[<set> x] [<set> y] [<set> z]] [implies [and [subset ?x ?y] [subset ?y ?z]] [subset ?x ?z]]]] are true. How can we get PLANNER to prove that [subset a c] holds? We would give the system the following theorems.

given:

```
        [subset a b]
        [subset a d]
        [subset b c]

<assert <define backward
        <consequent [[<set> x y z]] [subset ?x ?z]
                <unique>
                ;"the current goal must be unique"
                <goal
                        [subset ?x ?y]
                        [¬"use" ¬"current" backward <?>]>
                <goal
                        [subset .y ?z]
                        [¬"use" ¬"current" backward]>
                <assert [subset .x .z] [¬"try" <?>]>>>>
```

Now if we ask PLANNER to evaluate <goal [subset a c]> then we obtain the following protocol:

```
<goal [subset a c]>
        <current [subset a c]>
        fail
        <achieve [subset a c]>
        enter backward
        x becomes a
        z becomes c
        <unique>
        <goal [subset a ?y]>
                <current [subset a ?y]>
node 1,9
        y becomes d
        <goal [subset d c]>
                <current [subset d c]>
                fail
                <achieve [subset d c]>
                enter backward
                x becomes d
                z becomes c
                <unique>
                <goal [subset d ?y]>
                        <current [subset d ?y]>
                        fail
                        <achieve [subset d ?y]>
                        enter backward
                        x becomes d
                        z becomes ?y
                        <unique>
                        fail
                fail
node 1,9  ;note that this node appears above
        y becomes b
        <goal [subset b c]>
                <current [subset b c]>
        <assert [subset a c]>
        succeed
```

After the evaluation the data base contains:

```
        [subset a b]
        [subset a d]
        [subset b c]
        [subset a c]
```

In other words the first thing that PLANNER does is to look for a theorem that it can activate to work on the goal. It finds backward and binds x to a and z to c. Then it makes [subset a ?y] a subgoal with the recommendation that backward should be used first to try to achieve the subgoal. The system notices that y might be d, so it binds y to d. Next [subset d c] is made a subgoal with the recommendation that only backward be used to try to achieve it. Thus backward is called recursively, x is bound to d, and z is bound to c. The subgoal [subset d ?y] is established causing backward to again be called recursively with x bound to d and z determined to be the same as what the old value of y ever turns out to be. But now the system finds that it is in trouble because the new subgoal [subset d ?y] is the same as a subgoal on which it is already working. So it decides that it was a mistake to try to prove [subset d c] in the first place. Thus y is bound to b instead of d. Now the system sets up the subgoal [subset b c] which is established immediately. We use the above example only to show how the rules of the language work in a trivial case. If we were seriously interested in proving theorems in PLANNER about the lattice of sets, then we would construct a finite lattice as a model and use it to guide us in finding the proof.

5.4.2 Using an Antecedent Theorem

Suppose we give PLANNER only the following theorems.

```
given:
        [subset a b]
        [subset c d]

<assert <define forward-right
        <antecedent [[<set> x y z]] [subset _y _z]
                <goal [subset ?x .y]>
                <assert
```

```
                              [subset .x .z]
                              [¬"try" forward-right forward-left]>>>>

<assert <define forward-left
        <antecedent [[<set> x y z]] [subset _x _y]
                <goal [subset ?y .z]>
                <assert
                        [subset .x .z]
                        [¬"try" forward-right forward-left]>>>>
```

Now if PLANNER is asked to the theorem evaluate <assert [subset b c]

[¬"try" <?>]>, we obtain the following protocol:

```
<assert [subset b c]>
        <draw [subset b c]>
        enter forward-right
        y becomes b
        z becomes c
        <goal [subset ?x b]>
                <current [subset ?x b]>
        x becomes a
        <assert [subset a c]>
                <draw [subset a c]>
                enter forward-right
                y becomes a
                z becomes c
                <goal [subset ?x a]>
                        <current [subset ?x a]>
                        fail
                enter forward-left
                x becomes a
                z becomes c
                <goal [subset c ?z]>
                        <proved [subset c ?z]>
                z becomes d
                <assert [subset a d]>
                        <draw [subset a d]>
                        enter forward-right
                        y becomes a
                        z becomes d
                        <goal [subset ?x a]>
                                <current [subset ?x a]>
                                fail
                        enter forward-left
                        x becomes a
                        y becomes d
                        <goal [subset d ?z]>
                                <current [subset d ?z]>
                                fail
```

```
                    fail
                succeed
```

After the evaluation the data base contains:

```
        [subset a b]
        [subset c d]
        [subset a d]
        [subset b c]
        [subset a c]
```

Theorems in PLANNER can be proved in much the same way used for ordinary theorems. For example suppose that we had the following two theorems:

```
<assert <define th4 <consequent [[<set> a c]] [subset ?a ?c]
        <goal [set ?a]>
        <temprog [[<object> [x <arbitrary <object>>]]]
                <assert [element .x .a] <?>>
                <goal [element .x ?c]>>
        <assert [subset .a .c] <?>>>>>
```

The function ARBITRARY generates a unique symbol which has the type of its argument. On entrance to the function TEMPROG the identifier *x* is bound to a freshly created symbol. The above theorem is a constructive analogue of

```
[all [function <boole>[[<set> a] [<set> c]]
        [implies
                [all [function
                        <boole>
                        [[<object> x]]
                        [implies [element ?x ?a][element ?x ?c]]]
                [subset ?a ?c]]]]]
```

Going in the opposite direction, we have

```
<assert <define th4-5 <antecedent
        [[<set> a b]]
        [subset a b]
        <assert
            <antecedent
                [[<element> x]]
```

```
                    [element ?x ?a]
                    <assert [element ?x ?b] <?>> a>>>>>

<assert <define th4-6 <antecedent
        [[<set> a b]]
        [subset a b]
        <assert
            <consequent
                [[<element> x]]
                [element ?x ?b]
                <goal [element ?x ?a]> b>>>>>

<assert <define
        th3
        <consequent [[<object> x][<set> r s]]
                        [element ?x ?s]
                <goal [element ?x ?r]>
                <goal [subset ?r ?s]>
                <assert [element .x .s] <?>>>>>
```

The above theorem is a constructive analogue for

```
[all [function
        <boole>
        [[<object> x] [<set> s]]
        [implies
                [some [function
                        <boole>
                        [[<set> r]]
                        [and [element ?x ?r] [subset ?r ?s]]]
                [element ?x ?s]]]]]
```

From th3 and th3 we can prove the following theorem:

```
<consequent [[<set> a b  c]] [subset ?a ?c]
        <goal [subset ?a ?b]>
        <goal [subset .b ?c]>
        <assert [subset .a .c] <?>>>
```

The above theorem is a constructive analogue for

```
[all [function
        <boole>
        [[<set> a] [<set> b] [<set> c]]
        [implies
                [and [subset ?a ?b] [subset ?b ?c]]
                [subset ?a ?c]]]]
```

Often we treat the statement of a theorem simply as an abbreviation for the proof of the theorem.

We would like to examine the previous problem from the point of view of resolution based deductive system. The actor CLAUSE matchs clauses. It uses the fact that disjunction is commutative and associative. We have:

```
 1. <clause [[<set> a b] [<object> x]]
        [not [subset ?a ?b]]
        [not [element ?x ?a]]
        [element ?x ?b]>

 2. <clause [[<set> a b]]
        [element [element-of-difference ?a ?b] ?a]
        [subset ?a ?b]>

 3. <clause [[<set> a b]]
        [not [element [element-of-difference ?a ?b] ?b]]
        [subset ?a ?b]>

<assert <define necessary
  <antecedent
        [literal1 literal2]
        <clause <all [subset {?}] _literal1> <?>>
                <_
                        <clause
                                <all
                                        [not [subset {?}]]
                                        _literal2>
                                <?>>
                        <clause [[<set> a b] [<object> x]]
                                [not [subset .a .b]]
                                [not [element ?x .a]]
                                [element ?x .b]>>
                <assert <resolve .literal1 .literal2>>>>>
```

The above theorem says that we should eliminate all positive instances of the predicate subset from clauses. It is a special case of theorem1 which has been partially compiled.

```
<assert <define sufficient
  <antecedent
        [[<set> a b] literal1]
        <clause <all [not [subset _a _b]] _literal1> <?>>
        <prog [literal2]
                <_
                        <clause <all [subset (?} ] _literal2> <?>>
                        <clause [[<set> a b]]
                                [subset .a .b]
                                [element
                                        [element-of-difference .a .b]
                                        .a ]>>
                <assert <resolve .literal1 .literal2>>>
        <prog [literal2]
                <_
                        <clause <all [subset (?} ] _literal2> <?>>
                        <clause [[<set> a b]]
                                [subset .a .b]
                                [not [element
                                        [element-of-difference .a .b]
                                        .b]]>>
                <assert <resolve .literal1 .literal2>>>>>>
```

The above theorem says that we should eliminate all negative instances of the predicate subset from clauses.

5.4.3 Using Resolution

We shall assume that the resolution routines automatically detect contradictory pairs of clauses when they are generated. The theorem [implies [and [subset a b] [subset b c]] [subset a c]] can be proved as follows:

```
<prog [ ]
        <temprog [[<set>
                        [a <arbitrary <set>>]
                        [b <arbitrary <set>>]
                        [c <arbitrary <set>>]]]
                <assert <clause [ ] [subset .a .b]> [-"try" <?>]>
                <assert <clause [ ] [subset .b .c]> [-"try" <?>]>
                <assert <clause [ ] [not [subset .a .c]]> [-"try" <?>]>
```

```
                 <goal <clause>>>
         <assert <clause [[<set> x y z]]
                          [not [subset ?x ?y]]
                          [not [subset ?y ?z]]
                          [subset ?x ?z]>>>
```

The proof is:

```
4. <clause [ ]
       [subset a b]>

5. <clause  [[<set> x]]
       [not [element ?x a]] [element ?x b]> by 1. and 4.

6. <clause  [ ]
       [subset b c]>

7. <clause [[<set> x]]
       [not [element ?x _]] [element ?x c]> by 1. and 6.

8. <clause [ ]
       [not [subset a c]]>

9. <clause [ ]
       [element [element-of-difference a c] a]> by 8. and 2.

 10. <clause [ ]
       [element [element-of-difference a c] b]> by 8. and 3.

 11. <clause [ ]
       [not [element [element-of-difference a c] c]]> by 10. and 7.

 12. <clause [ ]
       [not [element [element-of-difference a c] b]]> by 9. and 5.

 13. <clause [ ]> by 12. and 10.
```

5.5 Myths about PLANNER

5.5.1 Consequent Theorems Are Used Only for Working Backwards

We would like to give an example to show that the computation tree that PLANNER defines as it executes theorems does not necessarily correspond to the tree of the intuitive solution space which is being investigated. The example which we use is the farmer, goat, cabbage, and wolf problem. We worked out the following solution with Jeff Rulifson. The problem begins with a farmer on the side of a stream with a boat, a wolf, a goat, and cabbage. The farmer wants to transport them all across the stream in the boat. The boat can only hold one of them besides the farmer. The wolf will eat the goat and the goat will eat the cabbage if the farmer is not there to interfere. How can the farmer get them all across the stream? We begin by evaluating <goal [from t t t t]> which means to set up a goal to make a move from the postion where all four objects are on the same side of the bank.

```
<assert <define make-move <consequent make
                  [wolf goat cabbage farmer]
        [from ?wolf ?goat ?cabbage ?farmer]
        <goal [safe .wolf .goat .cabbage .farmer]>
        ;"make sure the current situation is safe"
        <cond
            [<and?
                        <is? <> .wolf>
                        <is? <> .goat>
                        <is? <> .cabbage>
                        <is? <> .farmer>>
```

```
          <.make t>
          ;"exit .make with t"]>
;"if they are all safely on the other
          side of the river return t"
<cond
     [<current? [looked-at
                         .wolf
                         .goat
                         .cabbage
                         .farmer]>
          <.make <>>
          ;"exit .make with <>"]>
;"if we have already looked at this situation
          return <> which is false"
<assert [looked-at .wolf .goat .cabbage .farmer]>
<or
          &<cond
               [<is? .farmer .goat>
                    ;"if the farmer is on the same side
                         as the goat,
                         then he can carry the goat with
                         him to the other side"
                    <goal [from
                         .wolf
                         <not?  .goat>
                         .cabbage
                         <not? .farmer>]>]>
          &<goal  [from
                    .wolf
                    .goat
                    .cabbage
                    <not? .farmer>]>
          &<cond
               [<is? .farmer .wolf>
                    ;"similarly if the farmer is on the same side
                         as the wolf"
                    <goal [from
                         <not? .wolf>
                         .goat
                         .cabbage
                         <not? .farmer>]>]>
          &<cond
               [<is? .farmer .cabbage>
                    <goal [from
                         .wolf
                         .goat
                         <not? .cabbage>
                         <not? .farmer>]>]>>
;"the function OR tries the
          possibilities in order>>>
```

```
<assert <define safety-check <consequent safety-check
                   [wolf goat cabbage farmer]
        [safe ?wolf ?goat ?cabbage ?farmer]
        <cond
                   [<or?
                           <and?
                                   <is? .wolf <not? .farmer>>
                                   <is? .wolf .goat>>
                           <and?
                                   <is? .goat <not? .farmer>>
                                   <is? .goat .cabbage>>>
                   ;"the situation is not safe if either
                           the wolf is on the opposite side
                           from the farmer
                           but on the same side as the goat or
                           the goat is on the opposite side from the
                           farmer but on the same side as the cabbage"
                   <fail <> .safety-check>]>>>>
```

The protocol of the solution is:

```
<goal [from t t t t]>
    <goal [from t <> t <>]> goat
        <goal [from  t t t t]> goat
        <goal [from t <> t t]> himself
            <goal [from t <> t <>]> himself
            <goal [from <> <> t <>]> wolf
                <goal [from <> t t t]> goat
                    <goal [from <> <> t <>]> goat
                    <goal [from <> t t <>]> himself
                        <goal [from <> t t t]> himself
                        <goal [from t t t t]> wolf
                    <goal [from <> t <> <>]> cabbage
                        <goal [from <> t <> t]> himself
                                <goal [from <> <> <> <>]> goat
```

Note that there are several things wrong with the above procedure. For one thing the problem solver should work forwards and backwards simultaneously trying to find necessary conditions for a solution as well as sufficient condtions. The procedure is not very smart in the way that it goes about looking for a solution. These ills can be cured in various ways. The reader might find it instructive to consider some of the possibilities.

5.5.2 PLANNER Does only Depth First Search

PLANNER runs under a backtrack control structure. Because of the control structure the execution tree of a process looks like a depth first investigation. However, by creating more processes the growth of the set of execution trees can be quite arbitrary. As an example we can convert the above solution to the farmer, goat, cabbage, and wolf problem to breadth first investigation by evaluating the arguments to OR in parallel instead of sequentially in the theorem MAKE-MOVE.

5.5.3 Use of Failure Implies Inefficient Search

The failure primitive in PLANNER is a method of transferring control. The concept does not have any necessary relation to program errors such as dividing by zero. Often a proof by contradiction is completed by generating a failure back to an label function with a message like "happiness" when the contradiction is detected. The message is caught when it propagates back to the point where the proof by contradiction was set up. The effect of the failure is to get rid of all the garbage that is generated in the proof by contradiction. In a similar vein the failure mechanism is often used as a summarizing mechanism. At certain points along the computation, certain conclusions are derived from the process of investigation. These conclusions can be lifted out of the details that were used to derive them by failing back with values which summarize what has been learned. Then the computation can continue with a cleaner slate. For example in a chess program, exploration of the possible moves might reveal that our queen is pinned against our king threatening the loss of the queen. Information to that effect would be passed back with the failure.

5.5.4 PLANNER Does Only What It Is Told

In a strict sense PLANNER does only what it is told to do. There is no random element or independent consciousness built into the primitives. However, because of the goal oriented nature of the formalism it is very difficult to predict what a large body of PLANNER theorems will do. In fact one of the more obnoxious things that can happen is that some theorems find a nonobvious way to accomplish a trivial goal. Usually this happens because there is a bug in the code for the obvious way to achieve the goal.

6. More on PLANNER

6.1 PLANNER EXAMPLES

6.1.1 London's Bridge

Most of the time we decide which statements we want to erase on the basis of the justifications of the statements. If we erase statement a, and statement b depends on statement a because a is part of the justification of b, then we probably want to erase statement b. Sometimes a decision is made on the basis of other criteria. For example suppose that we carefully remove the bottom brick from a column of bricks. We shall suppose that each brick is of unit length. The statement [at |brick| |place| |height|] will be defined to mean that brick |brick| is at place |place| at the height |height|. Suppose that have the following theorems:

```
        [at brick1 here 0]
        [at brick2 here 1]
        [at brick3 here 2]

 <define london's-bridge
    <erasing
        [
                [<brick> brick other-brick]
                [<place> place]
                [<integer> height]]
        [at _brick _place _height]
                <erase
```

```
                    [at
                            _other-brick
                            .place
                            <add1 .height>]
                    [¬use¬ <?>]>
               ;"erase the fact that there is another brick
                    in the place above brick"
               <assert
                    [at .other-brick .place .height]>
               ;"assert that it is where
                    brick used to be">>
```

Thus after <erase [at brick1 here 0]> we will have [at brick2 here 0] and [at brick3 here 1]. The upper bricks in the tower have all fallen down one level. The above example comes from a suggestion made by S. Papert.

6.1.2 Analogies

6.1.2.1 Simple Analogies

Our next example illustrates the usefulness of the pattern directed deductive system that PLANNER uses compared with the quantificational calculus of order omega. We are interested in simple analogies such as those explored by Tom Evans. Given that object a1 has some relation to object a2 and that object c1 has the same relation to object c2, the problem is to deduce that a1 is analogous to c1. We use the predicate test-analogous within the theorem pair to record that we think two objects might be analogous and that we would like to check it out. Suppose that we give PLANNER the following theorems:

```
        [inside a1 a2]
        [inside c1 c2]
        [a-object a1]
        [a-object a2]
        [c-object c1]
        [c-object c2]

<define pair <consequent pair
        [
                [<object> a c]
                [<functor> predicate]
                [{?} argsa1 argsa2 argsc1 argsc2]]
        [analogous ?a ?c ?predicate]
                <unique>
                ;"the current goal must be unique"
                <cond
                    [<current? [test-analogous ?a ?c]>
                        ;"if a and c are test-analogous then
                                we are done"
                        <.pair done>
                        ;"exit .pair with done"]>
                <current [a-object ?a]>
                <current [c-object ?c]>
                ;"find an a-object and a c-object"
                <assert [test-analogous .a .c ?predicate]>
                <current [?predicate !_argsa1 .a !_argsa2]>
                <current [.predicate !_argsc1 .c !_argsc2]>
                ;"find a predicate in which both a and
                        c are arguments"
                <cond
                    [<is? <non [ ]> .argsa1>
                        <goal [corresponding-analogous
                                .argsa1
                                .argsc1
                                .predicate]>]>
                <cond
                    [<is? <non [ ]> .argsa2>
                        <goal [corresponding-analogous
                                .argsa2
                                .argsc2
                                .predicate]>]>
                ;"show that the other arguments are analogous"
                <assert [analogous .a .c .predicate]>>>


<define chop-off-another <consequent
        [
                [<object> a b]
                [{?} aa bb]
                [<functor <?> <?>> predicate]]
        [corresponding-analogous [?a ??aa] [?c ??cc] ?predicate]
                <cond
                    [<current? [test-analogous ?a ?c ?predicate]>
                        ;"if a and c are currently test-analogous then
                                we only have to look at
                                the rest of the elements"]
                    [¬"else"
                        <current [analogous ?a ?c ?predicate]>]>
                <cond
                    [<is? <non [ ]> .aa>
                         <current [corresponding-analogous
                                        ?aa
                                        ?cc
                                        ?predicate]>]>>>
```

Thus if we ask PLANNER to evaluate <goal [analogous a1 ?x inside]>

then x will be bound to c1 in accordance with the following protocol:

```
<goal [analogous a1 ?x inside]>
        enter pair
        a gets the value a1
        c gets the value ?x
        predicate gets the value inside
        <unique>
                <current [test-analogous a1 ?c inside]>
                FAIL
        <current [a-object a1]>
        <current [c-object ?c]>
node 1
        c gets the value c2
x gets the value c2
        <temporary [test-analogous a1 c2 inside]>
        <current [inside a1 a2]>
        <current [inside c1 c2]>
        <goal [corresponding-analogous [a2] [ ] inside]>
                enter chop-off-another
                FAIL
        FAIL
node 1; note that this node appears above
        c gets the value c1
x gets the value c1
        <temporary [test-analogous a1 c1 inside]>
        <current [inside c1 c2]>
        <goal [corresponding-analogous [a2] [c2] inside]>
                enter chop-off-another
```

```
a gets the value a2
c gets the value c2
       <current [test-analogous
                     a2
                     c2
                     inside]>
       FAIL
<current [analogous a2 c2]>
       enter pair
       a gets the value a2
       c gets the value c2
       <unique>
              <current [test-analogous
                            a2
                            c2
                            inside]>
              FAIL
       <current [a-object a2]>
       <current [c-object c2]>
       <temporary [test-analogous a2 c2 inside]>
       <current [inside a1 a2]>
       <current [inside c1 c2]>
       <goal [correspoinding-analogous
                     [a1]
                     [c1]
                     inside]>
              enter chop-off-another
              a gets the value a1
              c gets the value c1
              <current [test-analogous a1 c1]>
              succeed
```

In the process of carrying out the evaluation the following additional facts will be established: [analogous a1 c1 inside] and [analogous a2 c2 inside]. The reader might find it amusing to try to formulate the above problem in the first order quantificational calculus.

6.1.2.2 Structural Analogies

The process of finding analogous proofs and methods plays a very important role in theorem proving. For example the proofs of the uniqueness of the identity element and inverses in semi-groups are closely related. The definitions are:

[equivalent [identity e] [equal [* a e] [* e a] a]]
[implies [identity e] [equivalent [inverse b1 b] [equal [* b1 b] [* b b1] e]]] If e and e' are identities, then we have [equal e [* e e'] e']. If a1 and a1' are inverses of a, then we have [equal a1 [* a1' a a1] a1]. The general form of the analogy is [equal w _string w'] where .string algebraicly simplifies to w and w'. In many cases analogies are found by construction. That is the problem solver looks around for problems that might be solved with an analogous technique. In other words we will have a method of solution in search of a problem that it can solve! Now that we have found a technique for proving that various kinds of elements are unique, let us look around for a similar problem to which our technique applies. We find that zeros in semi-groups are defined as follows:
[equivalent [zero z] [equal [* a z] [* z a] z]] Supposing that z and z' are zeros we find that [equal z [* z z'] z']. One major problem in the effective use of analogies in order to solve problems is that it is very difficult to decide when and at what level of detail to try for an analogy. Another problem is that often the analogy holds only at a quite abstract level and it must not be pushed too far. Consider the following two algorithms:

```
<define number-of-atoms
  [function [x]
        <cond
```

```
            [<empty? .x>
                0]
            [<is? !=atom .x>
                1]
            [¬"else"
                <+
                        <number-of-atoms <1 .x>>
                        <number-of-atoms <rest .x>>>]>]>

<define list-of-atoms
  [function [x]
        <cond
            [<empty? .x>
                []]
            [<is? !=atom .x>
                [.x]]
            [¬"else"
                <append
                        <list-of-atoms <1 .x>>
                        <list-of-atoms <rest .x>>>]>]>
```

The functions number-of-atoms and list-of-atoms are precisely analogous. In most cases two functions will not be nearly so similar. Very few of the ideas of one will be used in the other. Structural analogies may also be constructed by procedural abstraction [see chapter 7]. Bledsoe has suggested that still another example of analogous proofs is found in the Schwartz inequality:

```
 <nondecreasing
        <expt
                <+
                        <* <x 1> <y 1>>
                        <* <x 2> <y 2>>>
                2>
        <*
                <+
                        <expt <x 1> 2>
                        <¬ <x 2> 2>>
                <+
                        <¬ <y 1> 2>
                        <¬ <y 2> 2>>>>
```

```
<nodecreasing
        <expt
                <sigma
                        1
                        n
                        <function
                                [i]
                                <expt <* <x .i> <y .i>> 2>>>
                2>
        <*
                <sigma
                        1
                        n
                        <function
                                [i]
                                <expt <x .i> 2>>>
                <sigma
                        1
                        n
                        <function
                                [i]
                                <expt <y .i> 2>>>>>

<nondecreasing
        <expt <integral <* x y>> 2>
        <*
                <integral <expt f 2>>
                <integral <¬ g 2>>>>
```

6.1.3 Mathematical Induction

We can formulate the principle of mathematical induction for the integers in the following way:

```
<define induction <consequent [p]
        [for-all _p]
                <temprog [[n <arbitrary <integer>>]]
                        <goal !'<.p 0>>
                        <assert !'<.p .n>>
                        <goal !'<.p !'<+ .n 1>>>>
                <assert [for-all .p]>>>
```

If we are given the facts [= <+ 0 0> 0] and

```
<clause [x y]
        [=
                <+ ?y <+ ?x 1>>
                <+ <+ ?y ?x> 1>]>
```

then we can establish

```
[for-all <function [n] [= <+ 0 ?n> ?n]>].
```

The following theorem will do induction on s-expressions:

```
<define expr-induction
<consequent
        [p]
        [for-all _p]
                <temprog
                                [[a <arbitary <atom>>]]
                        <goal !'<.p .a>>>
                <temprog
                                [
                                [car <arbitrary <expr>>]
                                [cdr <arbitrary <expr>>]]
                        <assert !'<.p .car>>
                        <assert !'<.p .cdr>>
                        <goal !'<.p !'<cons .car .cdr>>>>
                <assert [for-all .p]>>>
```

We would like to try to do without existential quantifiers. We can eliminate them in favor of Skolem functions in assertions and in favor of PLANNER identifiers in goals. The problem of finding proofs by induction is formally identical to the problem of syntesizing programs out of "canned loops". The process of procedural abstraction [which is explained in chapter 7] has an analogue which is "induction abstraction" [finding proofs by induction from example proofs written out in full without induction].

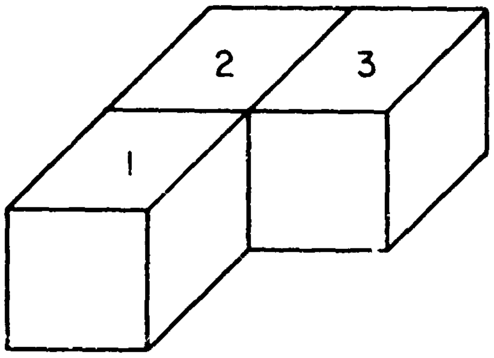
6.1.4 Descriptions

6.1.4.1 Structural Descriptions

PLANNER can be used to find objects from partial or schematic descriptions. The statement [perpendicular [line _a _b] [line _c _d]] will be defined tc mean that the lines [line .a .b] and [line .c .d] are perpendicular. The MATCHLESS function <ASSIGNED? arg> tests to see if the identifier arg has a value. We shall adopt the convention that [glued a b] means that bricks a and b are glued together and [orthogonal [line |a| |b|] [line |c| |d|]] means that the lines between the centers of bricks |a| and |b| is orthoganal to the line between the centers of bricks |c| and |d|. A three-corner is defined to be a group of three bricks joined together such that two of them are diagonal to each other. A three-corner is shown in figure 1. In other words the following is a description of a three-corner:

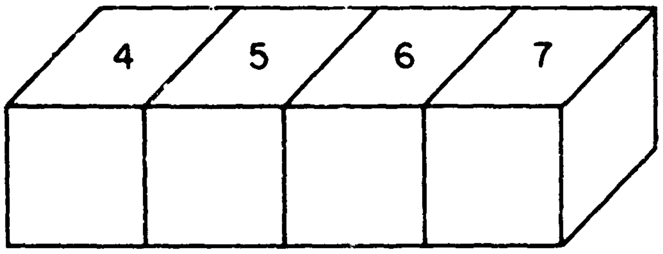
```
<define find-three-corner
 <consequent
     [[<brick> a b c]]
     [three-corner ?a ?b ?c]
        <goal [glued ?a ?b]>
        <prog again [ ]
                 <goal [glued
                                 .a
                                 <all <non .b> ?c>]>
                  <goal [orthogonal [line .a .b] [line .a .c]]>
                  <cond
                    [<or?
                            &<goal [glued
```

6.1 page 265



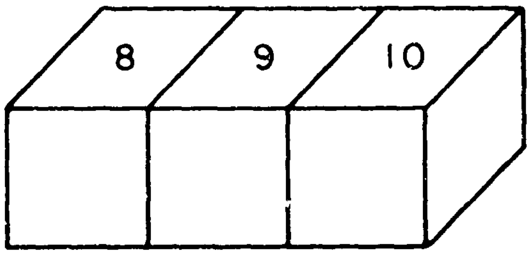
A Three-Corner:

[cube 1]	[glued 1 2]
[cube 2]	[glued 2 3]
[cube 3]	



A STICK:

[cube 4]	[glued 4 5]
[cube 5]	[glued 5 6]
[cube 6]	[glued 6 7]
[cube 7]	



ANOTHER STICK:

[cube 8]	[glued 8 9]
[cube 9]	[glued 9 10]
[cube 10]	

```
                                        .a
                                        <all <non .b> <non .c>>]>
                              &<goal [glued .b <non .a>]>
                              &<goal [glued .c <non .a>]>>
                          <fail <> again>]>>>>
```

The description can be used in the obvious way to find three-corners.

The statement [stick _a _b] is defined to mean that .a and .b are end bricks of a line of bricks and [between _a _b _c] is defined to mean that brick .b is between bricks .a and .c. Examples of sticks are shown are shown in figure 1.

```
<define find-stick
         <consequent
                [[<brick> a b] [!=fix n]]
                [stick ?a ?b _n]
                        <current [brick ?a]>
                        <current [brick ?b]>
                        <goal [stick-segment .a .b <- .n 2>]>
                        <assert [stick .a .b .n]>>>

<define find-stick-segment
 <consequent find
 [[<brick> x y w][!=fix n]]
 [stick-segment ?x ?y _n]
        <cond
            [<is? <neg> .n>
                <fail>]
            [&<goal [glued ?w ?x]>
                <goal [orthogonal
                        [line .x .w]
                        [line .x ?y]]>
                <fail>]
            [&<goal [glued ?x ?y]>
                <cond
                    [<and?
                                <goal [glued ?w ?y]>
                                <goal [orthogonal
                                        [line .y .w]
                                        [line .y .x]]>>
                        <fail>]>
                <.find t>
                ;"exit .find with t"]>
        <goal [glued ?w .x]>
        <goal [between .x .w .y]>
        <goal
                [stick-segment .w .y <- .n 1>]
                [¬use¬ find-stick-segment <?>]>>>
```

6.1.4.2 Constructing Examples of Descriptions

Given a description of a structure [such as a stick] we would like to be able to derive a general method for building the structure. The problem of deriving such general construction methods from descriptions is very difficult. In this case we we can construct a stick of length n with ends x and y using the functions <GLUE face1 face2> which glues the value of face1 to the value of face2 and the function new-brick which produces a new brick.

```
<define make-stick <consequent make
        [[<brick> x y w] [!=fix n]]
        [make-stick _x _y _n]
                <cond
                    [<is? <less 3> .n>
                        <glue [bottom .x] [top .y]>
                        <.make t>
                        ;"exit .make with t"]>
                <is _w <new-brick>>
                <glue [bottom .x] [top .w]>
                <goal [make-stick _w _y <- .n 1>]>>>
```

6.1.4.3 Descriptions of Scenes

S. Papert has suggested that theorem proving techniques might be applied to the problem of analyzing 2-dimensional projections of 3-dimensional bricks. In this section we will give a formal definition of the problem. Adolpho Guzman has developed a program [called SEE] which tries to solve such problems Many humans solve such problems by mentally constructiing a symbolic 3-dimensional scene which optically projects back to the given 2-dimensional input. We define a brick to be a connected open opaque region of 3-space bounded by a finite number of planes such that if two planes intersect then they must be orthogonal. Furthermore the complement of a brick is required to be connected. Thus bricks are allowed to have holes in them. A 3-dimensional scene is an arrangement of bricks such that no two of them intersect. A 2-dimensional scene is a collection of straight lines in a plane. A 2-dimensional projection is the optical projection of a 3-dimensional scene onto a plane. A statement p about 3-dimensional scenes will be said to be valid for a 2-dimensional scene r if for all 3-dimensional scenes t such that t projects to r it is the case that p is true for t. A two dimensional scene r0 will be said to be ambiguous for a language l if it is the projection of two 3-dimensional scenes t1 and t2 such that there is a sentence p0 in l with p0 true in t1 and false in t2. There are a number of primitive predicates that should be included in a language for scene analysis:

[parallel x y] means that x and y are parallel.

[coplanar x y] means that x and y are coplanar.

[normal plane1 directed-linesegment] means that the normal of plane1 is in the direction of the directed-linesegment.

[restricted plane1 pt1 pt2 pt3] means that the normal to plane1 is restricted to the angle pt1 pt2 pt3.

[same-brick region1 region2] means that region1 and region2 are part of the same brick.

[adjacent region1 region2] means that region1 and region2 are regions of the same brick that intersect at right angles.

[convex region1 region2] means that region1 and region2 are regions of the same brick that intersect at right angles to make a convex body.

[concave region1 region2] means that region1 and region2 are regions of the same brick that intersect at right angles to make a concave body.

[element x y] means that x is an element of y.

[in-front-of brick1 brick2] means that brick1 is in front of brick2.

[resting-on brick1 brick2] means that brick1 is resting on brick2.

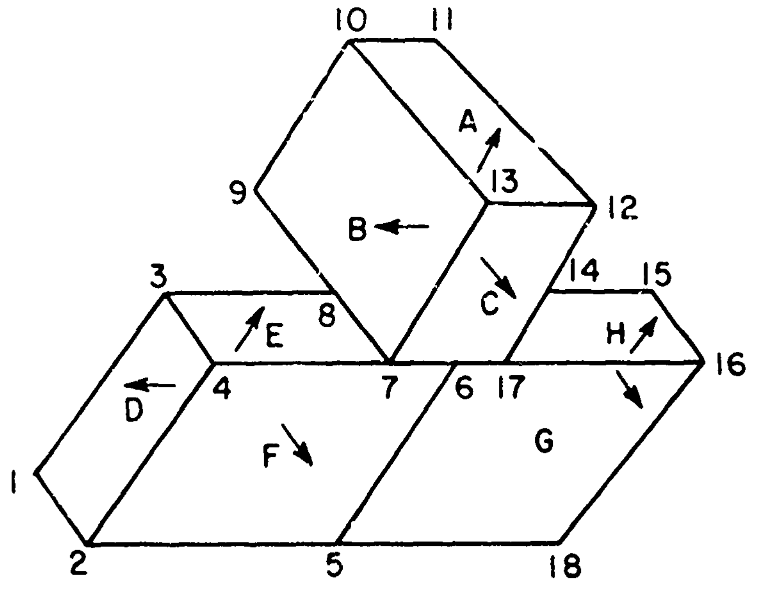
[on-top-of brick1 brick2] means that brick1 is on top of brick2.

[subset x y] means that x is a subset of y.

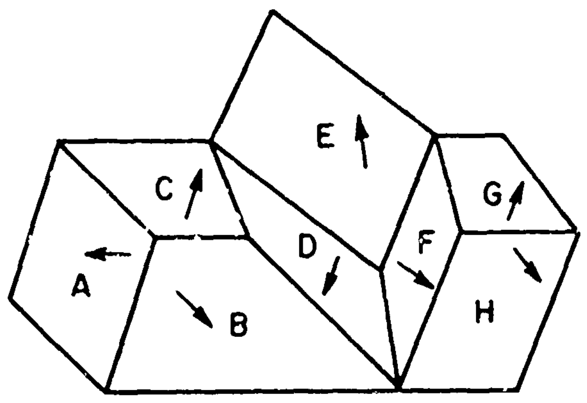
[coordinates point1 coord1] means that point1 has 3-dimensional coordinates coord1.

The following statements about example1 are valid as can be seen by considering where the normals of the planes might lie and deducing consequences until contradictions are found.

```
[normal a [direction 7 13]]
[normal b [direction 12 13]]
[convex a b]
```



EXAMPLE 1



EXAMPLE 2

```
[convex a c]
[convex b c]
[normal c [direction 10 13]]
[normal d [direction 7 4]]
[normal e [direction 2 4]]
[convex d e]
[normal f [direction 3 4]]
[convex d f]
[convex e f]
[normal h [direction 16 18]]
[normal g [direction 15 16]]
[convex g h]
```

The following statement about example 1 satisfiable:

```
[and
      [resting-on [brick a b c] [brick e f d]]
      [resting-on [brick a b c] [brick g h]]]
```

The following statements about example 2 are valid:

```
[convex a c]
[convex a b]
[convex b c]
[normal a [direction 12 14]]
[normal c [direction 3 14]]
[convex g h]
[normal g [direction 5 6]]
[normal h [direction 8 6]]
[not [adjacent  c d]]
[not [adjacent b d]]
[convex d e]
[convex e f]
[convex d f]
[normal e [direction 4 13]]
[normal d [direction 9 13]]
[normal f [direction 11 13]]
```

The following statement about example 2 is satisfiable:

```
[and
      [same-region  c  g]
       [same-region  b  h]
       [same-brick  a  b  c  g  h]]
```

The three dimensional coordinates of points are obtained by using more than one camera to view the scene or using a focus map. In the case where we have coordinates as a primitive predicate, the definition of a projection of a 3-dimensional scene must be modified to include the 3-dimensional coordinates of all the projected vertices. In the case where we have the three dimensional coordinates of the projected vertices, we can deduce that two planes are part of the same brick if they intersect at an acute right angle. Since the object that is being viewed might be so far away that accurate coordinates cannot be obtained, a deductive system should be developed which does not use coordinates. At the very minimum a hard core deductive system for the analysis of 2-dimensional projections should be consistent and every valid statement should be proveable. That is every theorem of the system should be satisfiable [there is at least one interpretation that satisfies the theorem]. Interest in questions of satisfiability comes from the fact that some interpretations are far more likely than others in the real world. Statements that are to be tested for satisfiability must be made as strong as possible in order to provide a meaningful test. Although the linking rules are mathematically very elegant, in their present form they do not adequately represent the semantics of the optical projection rules. The value of the program by Guzman is that it provides conjectures about which regions are satisfiable in the relation same-brick. However, the program suffers because it does not have any explicit knowledge of optics. We would advocate an approach that makes greater use of deduction to test the validity or satisfiability of a sentence. Questions of satisfiability and validity of sentences with respect to any given projection are decidable since the theory of real closed fields is decidable. Efficient algorithems should be developed to test whether a given sentence is valid or satisfiable in a projection.

6.1.4.4 Power Set of Intersection of Two Sets Is the Intersection of Their Power Sets

The following example was proposed by W. Bledsoe. Prove that the power set of the intersection of two sets is the intersection of their power sets. We shall use cap as a synonym for intersection.

```
<define extensionality-conse <consequent [[<set> x y]]
        [= ?x ?y]
        <goal [subset ?x ?y]>
        <goal [subset ?y ?x]>
        <assert [= .x .y]>>>

<define element-power-conse <consequent [[<set> x a]]
        [element ?x [power ?a]]
        <goal [subset ?x ?a]>
        <assert [element .x [power .a]]>>>

<define element-power-ant <antecedent [[<set> x a]]
        [element ?x [power ?a]]
        <assert [subset ?x ?a]>>>

<define subset-cap-conse <consequent [[<set> a b c]]
        [subset ?c [cap ?a ?b]]
        <goal [subset ?c ?a]>
        <goal [subset ?c ?b]>
        <assert [subset .c [cap .a .b]]>>>

<define subset-cap-ant <antecedent [[<set> a b c]]
        [subset _c [cap _a _b]]
        <assert [subset .c .a]>
        <assert [subset .c .b]>>>

<define subset-cap-conse <consequent [[<set> a b c]]
        [subset ?c [cap ?a ?b]]
        <goal [subset ?c ?a]>
        <goal [subset ?c ?b]>
        <assert [subset .c [cap .a .b]]>>>

<define element-cap-ant <antecedent [x [<set> a b]]
        [element _x [cap _a _b]]
        <assert [element .x .a]>
        <assert [element .x .b]>>>

<define element-cap-conse <consequent [x [<set> a b]]
        [element ?x [cap ?a ?b]]
        <goal [element ?x ?a]>
        <goal [element ?a ?b]>
        <assert [element ?x [cap ?a ?b]]>>>

<define subset-conse <consequent [[<set> a b]]
        [subset _a ?b]
        <temprog [x <arbitrary <?>>]
                <assert [element .x .a]>
                <goal [element .x .b]>>
        <assert [subset .a .b]>>>
```

We can now set up our goal to prove the theorem:

```
<goal [=
        [cap [power a1] [power a2]]
        [power [cap a1 a2]]]>
```

The goal will produce the following protocol:

```
enter extensionality-conse
        x becomes [cap [power a1] [power a2]]
        y becomses [power [cap a1 a2]]
        <goal [subset [cap [power a1] [power a2]][power [cap a1 a2]]]>
        enter subset-conse
                a becomes [cap [power a1] [power a2]]
                b becomes [power [cap a1 a2]]
                x becomes g1
                <assert
                        [element
                                g1
                                [cap [power a1] [power a2]]]>
                enter element-cap-ant
                        x becomes g1
                a becomes [power a1]
                b becomes [power a2]
                <assert [element g1 [power a1]]>
                enter element-power-ant
                        <assert [subset g1 a1]>
                <assert [element g1 [power a2]]>
                enter element-power-ant
                        <assert [subset g1 a2]>
        <goal [element g1 [power [cap a1 a2]]]>
        enter element-power-conse
                x becomes g1
                a becomes [cap a1 a2]
                <goal [subset g1 [cap a1 a2]]>
                enter subset-cap-conse
                        c becomes g1
                        a becomes a1
                        b becomes a2
                        <goal [subset g1 a1]>
                        <goal [subset g1 a2]>
                        <assert
                                [subset
                                        g1
                                        [cap a1 a2]]>
        <assert
                [element
                        g1
                        [power [cap a1 a2]]]>
<assert
        [subset
                [cap [power a1] [power a2]]
                [power [cap a1 a2]]]>
<goal
        [subset
                [power [cap a1 a2]]
                [cap [power a1] [power a2]]]>
enter subset-conse
        a becomes [power [cap a1 a2]]
        b becomes [cap [power a1] [power a2]]
        x becomes g2
        <assert [element g2 [power [cap a1 a2]]]>
        enter element-power-ant
                x becomes g2
                a becomes [cap a1 a2]
                <assert [subset g2 [cap a1 a2]]>
                enter subset-cap-ant
                        x becomes g2
                        a becomes a1
                        b becomes a2
                        <assert [subset g2 a1]>
                        <assert [subset g2 a2]>
        <goal
                [element
                        g2
                        [cap [power a1] [power a2]]]>
        enter element-cap-conse
                x becomes g2
                a becomes [power a1]
                b becomes [power a2]
                <goal [element g2 [power a1]]>
                enter element-power-conse
                        x becomes g2
                        a becomes a1
                        <goal [subset g2 a1]>
                <goal [element g2 [power a2]]>
                enter element-power-conse
                        x becomes g2
                        a becomes a1
                        <goal [subset g2a1]>
                <assert
                        [element
                                g2
                                [cap [power a1] [power a2]]]>
        <assert
                [subset
                        [power [cap a1 a2]]
                        [cap [power a1] [power a2]]]>
<assert
        [-
                [power [cap a1 a2]]
                [cap [power a1] [power a2]]]>
```

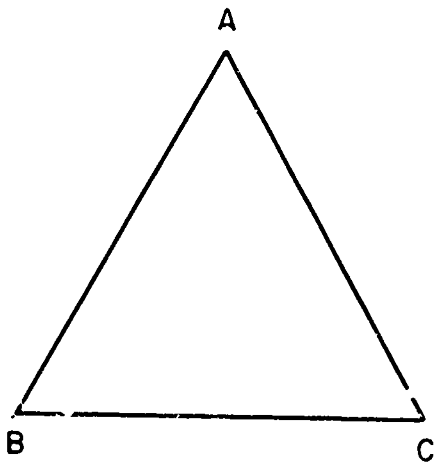
6.1.5 Semantics of Natural Language

Although problems for PLANNER are typically phrased in a perfectly formal, precise, unambiguous syntax, we will usually not find the semantics as well defined. If we say [[very happy] john] instead of "John is very happy." we will not thereby have made the concept of happiness any less nebulous for the machine. Nevertheless it is convenient for a problem solver to have such concepts although they are not rigorously defined. Problems of semantic ambiguity and clarificaton can require arbitrary amounts of computation in order to be adequately resolved. For example consider the following simple example of how semantic ambiguities can be eliminated with the aid of "real-world" knowledge:

```
        <assert [is-smaller-than hand [pig pen]]>

    <assert
        <define example-of-bar-hillel
           <antecedent [[<object> x y]]
                [in _x _y]
                <cond
                   [<is? pen .x>
                        <goal [is-smaller-than ?y [pig pen]]>
                        <assert [in [fountain pen] .y]>]>>>>
```

Now if we assert [in pen hand], PLANNER will conclude that [in [fountain pen] hand] is true since a hand is smaller than a pig pen. One of the important difficulties that have plagued most of the programs that have been written to answer questions in English is that they are trying to solve two very hard problems at the same time. First they must make sense of English syntax and second they need a powerful problem solving capability to answer the question once they have "understood" it. Ambiguous cases should be resolved on the basis of deduction and not on the basis of some linking scheme such as "semantic memory". As it stands PLANNER provides sophisticated mechanisms for solving problems in formal languages. A program could be written [perhaps in PLANNER?] to translate English into PLANNER theorems for problem solving. Conversely we could try to translate PLANNER theorems into simple natural language. Surprisingly translation into natural language can be very awkward because natural

THE PONS ASINORUM



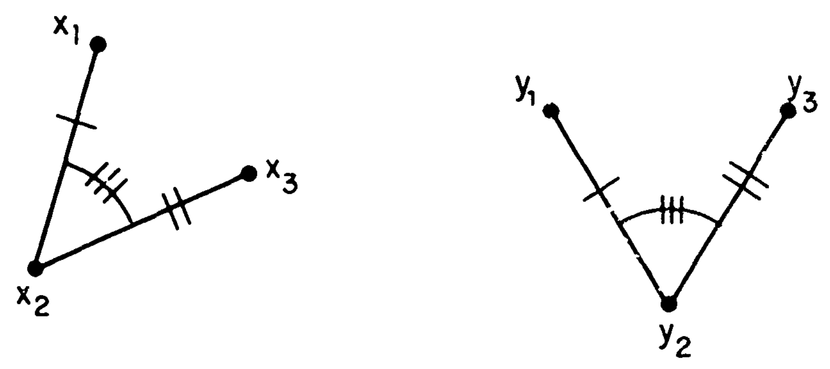
GIVEN : AB = AC

PROVE : $\angle ABC = \angle ACB$

6.1 page 277b

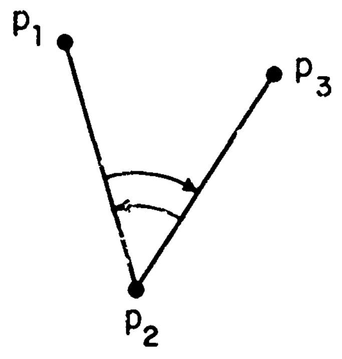
DIAGRAMS FOR GEOMETRY THEOREMS

SIDE-ANGLE-SIDE



[CONGRUENT [x_1 x_2 x_3] [y_3 y_2 y_1]]

EQUAL-ANGLE



[EQUAL [ANGLE p_1 p_2 p_3] [ANGLE p_3 p_2 p_1]]

language lacks many of the descriptive and procedural primitives of PLANNER.

6.1.6 The Pons Asinorum

We would like to show how the "bewilderingly simple" proof of the pons asinorum [i. e., base angles of an isosceles triangle are equal] can be done very simply in PLANNER. The following notation will be used:

[length |p1| |p2|] for the length from point |p1| to |p2|

[angle |x| |y| |z|] for the angle |x| |y| |z| which has the point |y| at its vertex

Four PLANNER theorems are used. They are procedural analogues of axioms in plane Euclidean geometry.

```
<define side-angle-side
    <consequent [x1 x2 x3 y1 y2 y3]
        [congruent [?x1 ?x2 ?x3] [?y1 ?y2 ?y3]]
                <unique>
                <goal [= [length ?x1 ?x2] [length ?y1 ?y2]]>
                <goal [= [angle ?x1 ?x2 ?x3] [angle ?y1 ?y2 ?y3]]>
                <goal [= [length ?x2 ?x3] [length ?y2 ?y3]]>>>

<define equal-angle
    <consequent [p1 p2 p3 w]
        [= [angle ?p1 ?p2 ?p3] ?w]
                <unique>
                <goal [= [angle ?p3 ?p2 ?p1] ?w]>>>

<define equal
    <consequent [x y]
        [= ?x ?y]
                <unique>
                <or
                        <match ?x ?y>
                        <goal [= ?y ?x]>>>>
```

```
<define angles-by-congruence
    <consequent [p1 p2 p3 q1 q2 q3]
        [= [angle ?p1 ?p2 ?p3] [angle ?q1 ?q2 ?q3]]
                <unique>
                <goal
                        [congruent
                                [?p1 ?p2 ?p3]
                                [?q1 ?q2 ?q3]]>>>
```

Suppose that we have an isosceles triangle ABC with the length of AB equal to the length of AC. We can input this as:

```
        <assert [= [length A B] [length A C]]>
```

The goal is to prove that angle ABC is equal to angle ACB:

```
<goal [= [angle A B C] [angle A C B]]>
```

One protocol for establishing the goal is:

```
    enter angle-by-congruerce
    p1 becomes A
    p2 becomes B
    p3 becomes C
    q1 becomes A
    q2 becomes C
    q3 becomes B
    <goal [congruent [A B C] [A C B]]>
        enter side-angle-side
        p1 becomes A
        p2 becomes B
        p3 becomes C
        q1 becomes A
        q2 becomes C
        q3 becomes B
        <goal [= [length A B] [length A C]]> is easy since it is in
the data base
        <goal [= [angle B A C] [angle C A B]]>
            enter equal-angle
            p1 becomes B
            p2 becomes A
            p3 becomes C
            w becomes [angle C A B]
            <goal [= [angle C A B]  agle C A B]]>
                enter equal
                x becomes [angle C A B]
                y becomes [angle C A B]
        <goal [= [length A C] [length A B]]>
            enter equal
            x becomes [length A C]
            y becomes [length A B]
                <goal [= [length A B] [length A C]]> succeeds by
looking in the data base
```

Ira Goldstein has implemented a Gerlernter-like geometry theorem prover.

6.2 Current Problems and Future Work

PLANNER would benefit greatly from an efficient parallel processing capability. The system would run faster if it could work on its goals in parallel. Quite often a goal will fail after a short computation along its path. The use of parallelism would enable us to get many goals to fail so that we could adopt more of a progressive refinement strategy. We would like to carry out computations to try to reject a proposed subgoal at the same time that we are trying to satisfy it. Many computations can be carried out much faster in parallel than in serial. For example we can determine whether a graph with n nodes is connected or not in a time proportional to <* <log n> <log n>>. It has been known for a long time that LISP computations using parallel evaluation of arguments are determinate if the functions rplaca, rplacd, and setq are prohibited. We could impose a similar set of restrictions on PLANNER. Another approach is to introduce explicit parallelism into the control structure. We have "|<" and ">" delimit parallel calls for elements and "|{" and "}" delimit parallel calls for segements. A parallel function call will act as a fork in which one process is created to do the function call and the other proceeds with normal order evaluation. For example in <+ |<* 3 4> <+ 7 8>> we could compute 3*4 in parallel with 7+8. The copy function could be sped up by a factor proportional to the number of processors:

```
<define copy [function [x]
        <cond
            [<is? <monadic> .x>
                .x]
           [¬"else"
                [!<copy <1 .x>> !<copy <rest .x>}]]>]>
```

However, we would still have problems communicating between the branches of the computation proceeding in parallel. Partly this a problem of sharing an indexed global data base between parallel processes. We would need the standard lock and unlock primitives and unlimited use of assignment in order to keep the computations synchronized. But if we allowed the use of lock and unlock and unlimited use of assignment, the programs might become indeterminate. One of the most important properties that can be proved about a program is that it is determinate.

PLANNER logic is a kind of hybrid between the classical logics [such as the quantificational calculus and intuitionistic logic], and the recursive functions [as represented by the lambda calculus and Post productions]. The semantics of PLANNER logic is most naturally defined dynamically by the properties of procedures. The semantics of the quantificational calculus can be defined by set theoretic models of possible worlds. The logic of the quantificational calculus is CONSERVATIVE in the sense that if a sentence S follows from a set of sentences M then S will follow from any superset of M. Do to its ability to have conditional expressions that test the state of the world, PLANNER logic is NOT conservative. This causes consternation among classical logicians because many elegant theorems for classical logic do not hold for PLANNER logic. The restriction of having to be conservative is quite severe in problem solving. Suppose that there are three cubes A, B, and C sitting on a table. Suppose that it is desired to build a tower two cubes high at place P. The plan constructed might be to pick up A, set it down at P, and then place B on top of it. If in the process of constructing the plan we deduced that cube A was glued to the table with liquid iron, we would want to change our plan to use cubes B and C to make the tower. But by the conservative properties of ordinary logic the original plan must remain valid. The only way around this would appear to be introduce some special kind of internal state into the deductive machinery of the quantificational calculus. Recommendations are another source of nonconservative behavior in PLANNER. For example we might not allow Zorn's Lemma to be used more than once in a proof. Both PLANNER logic and quantificational logic are COMPACT in the sense that a computation [proof] depends on only a finite number of expressions. In comparison with the quantificational calculus PLANNER would appear to be more powerful in the following areas:

control structure
pattern matching
erasure
local states of world

There are interesting parallels between theorem proving and algebraic manipulation. The two fields face similar problems on the issues of simplification, equivalence of expressions, intermediate expression bulge, and man-machine interaction. The parallel extends to the trade off between domain dependent knowledge and efficiency. In any particular case, the theorems need not allow PLANNER to lapse into its default conditions. It will sometimes happen that the heuristics for a problem are very good and that the proof proceeds smoothly until almost the very end. Then the progam gets stuck and lapses into default conditions to try to push through the proof. On the other hand the program might grope for a while trying to get started and then latch onto a theorem that knows how to polish off the problem in a lengthy but fool proof computation. PLANNER is designed for use where one has great number of interrelated procedures [theorems] that might be of use in solving some problem along with a general plan for the solution of the problem. The language helps to select procedures to refine the plan and to sequence through these procedures in a flexible way in case everything does not go exactly according to the plan. The fact that PLANNER is phrased in the form of a language forces us to think more systematically about the primitives needed for problem solving. We do not believe that computers will be able to prove deep mathematical theorems without the use of a powerful control structure. Nor do we believe that computers can solve difficult problems where their domain dependent knowledge is limited to finite-state difference tables of connections between goals and methods. Difference tables can be trivially simulated by conditional expressions in PLANNER.

Difficult problems for PLANNER

We would be grateful to any reader who could suggest types of problems which might be difficult to encompass naturally within the present formalism. PLANNER is intended to be a good language for the creation and description of problem solving strategies. Currently it operates within the restriction of generalized stack discipline. By relaxing this restriction we could make the language completely restartable at the considerable cost in efficiency of having to garbage collect the stack.

Speed: PLANNER runs best on a fast general purpose computer. However two special kinds of hardware would be useful. Alan Kay has pointed out that special hash code hardware could make the functions GET and PUT as fast for nodes as indexing hardware does for vectors. Second if we had a load thru mask instruction, then we could speed up monitoring. The instruction would interrupt if the appropriate monitor bits were on. Both of the above kinds of instructions should probably be micro-coded.

Memory: There is never enough fast random access storage. Furthermore the eighteen bit address space of the PDP-10 is inadequate. We need a bigger address space for the following purposes:

Garbage collection

Breathing space between data spaces [especially stacks]

Backtracking

Dynamic linking

Exploding definitions: We cannot afford to replace every term by its definition in trying to prove theorems. However, in the proof of almost every theorem it is necessary to replace some terms by their definitions. Domain dependent methods must be developed to make the decision in each case.

Creating PLANNER theorems: We need to determine when it is desireable to construct PLANNER theorems as opposed to dynamically linking them together at run time. At the present we have only a few examples of nontrivial constructed theorems. We can generate some from the functional abstraction of protocols and from attempts to construct schematic proofs of theorems. Others are generated as the answers to simple problems. For example if we ask the computer how it would put all the small green and yellow bricks in the red box, then it might answer:

```
<for [[<face> face1 face2] [<brick> brick]]
                [[¬"current" [small-brick _brick]]]
     <current [face _face1 .brick]>
     <current [color .face1 green]>
     <current [face _face2 .brick]>
     <current [color .face2 yellow]>
     <pi  -up .brick>
     <carry-to [above [red box]]>
     <drop>>
```

Terry Winograd has developed a program to translate English into PLANNER theorems. An interesting exprериment that could be attempted would be to modify a chess porgram so that it would return a PLANNER program as well as the symbolic description of a position. The idea is that the PLANNER program would represent the plan of action that would be taken in case of the various moves that the opponent might take. William Henneman has investigated some of the possibilites for doing planning in king and pawn end games. The problem seems to be very difficult but not impossible given the present state of the art.

Arbitrary Constraints: Using procedures as a semantic base requires us to solve the problem of making procedural formalisms more goal-oriented. The quantificational calculus is very goal oriented but suffers growing pains trying to introduce procedural knowledge.

Manipulation of PLANNER theorems: PLANNER provides a flexible computational base for manipulating theorems that can be put in disjunctive normal form. We need to deepen our understanding so that we can carry out similar manipulations on PLANNER theorems with the same facility.

Progressive refinement: We need to make more use of the style of reasoning in which we construct a plan for the solution of a problem from necessary conditions that the solution must have, attempt to execute the plan, find out why it does not work, and then try again. The style is often used in chess where very much the same game tree is gone over several times; each time with a deeper understanding of what factors are relevant to the solution.

Garbage collection of assertions: Statements which have been asserted should go away automatically when they can no longer be of use. Unfortunately, because of some logical problems and becuause of the retrieval system of PLANNER, we have difficulty in achieving completely automatic garbage collection. The erase primitive of the language provides one way to get rid of unwanted statements. If the asserted statement appears in the local state of some process instead of in the global data base then it will disappear automatically.

Simultaneous goals: We often find that we need to satisfy several goals simultaneously. We usually try to accomplish this by choosing one of the goals to try to achieve first. However, when working on the goal, we should keep in mind the other constraints that the goal must satisfy. One solution is to pass the goal to be worked on as a list whose first element is the goal and whose succeeding elements are the other goals which must be simultaneously satisfied.

Nonconstructive proofs: The most natural way to do a proof by contradiction is to try to calculate in advance the statement which ultimately will produce the contradiction. The method is to find a statement S such that S is provable and [not S] is provable. More precisely, we compute a statement S, make S a goal, and then make [not S] a goal. Bob Boyer has pointed out that in mathematics if the goal is to prove S, then if at any point in the proof the main goal reduces to the subgoal to prove [not S], then a proof by contradiction can be completed.

Models of Domains: Suppose that M is model for the set of hypotheses H with consequent C. Using constructive logic a subgoal S of the goal C would be rejected if it could be shown that it was unsatisfiable by M. Often rejections are made on the basis of a model. For example in the intuitive model of Zermelo-Fraenkel set theory all the descending element chains are finite and terminate in the null set. Furthermore every set has an ordinal rank. Thus the ordinals form the back bone of the set theory. The intuitive meaning of [+ A B] [where A and B are ordinals] is the concatenation of A with B. The intuitive meaning of [* A B] is the concatenation of A with itself B times. If two ordinals have the same order type then they are equal. Thus intuitively we would expect that [= [+ 1 omega] omega] is true. Every well developed mathematical domain is built around a complex of intuitive models and simple examples and procedures. Axiom sets are constructed to attempt to rigorously capture and delineate various parts of the complex. One of the most important criteria for judging the importance of a theorem is the extent to which it sheds light on the complex of the domain. These complexes must be mechanized. We conclude that it is unlikely that deep mathematical theorems can be proved solely from axioms and definitions by a uniform proof procedure. A uniform proof procedure based on model resolution does not provide the means for mechanizing the complex of a domain. Model resolution is a strategy for deciding which clauses to resolve. There is a great deal more to mechanizing the complex of a domain than simply pruning proof trees. Furthermore, clauses are often false in a model even though they are irrelevant to the proof that is being sought. One way that is often used to try to find a counterexample to a false statement about ordinals is to attempt to construct the counterexample from well known ordinals. Some well known ordinals are 1, 2, 3, omega, the least uncountable ordinal, etc. Thus in seeking a counter example to the statement that there are only finitely many limit ordinals less than a given ordinal we need go no further than [* omega omega].

7. Models of Procedures and the Teaching of Procedures

7.1 Models of Procedures

7.1.1 Models of Expressions: Intentions in INTENDER

A problem solver needs to have some way to know the properties of the procedures which it uses to solve problems. It can use the knowledge which it has as a partial model of itself. In order to be able to model procedures, it needs:

1: a way to express properties of procedures.

2: A way to establish that the properties do in fact hold for the procedures.

INTENDER is a goal-oriented formalism for expressiong models of procedures. The models are expressed in terms of intentions of what the procedure should accomplish. The primitives of INTENDER are concerned with expressing intentions in procedural terms. Thus the intentions are capable of themselves having intentions. INTENDER mechanizes the knowledge needed to do execution induction on procedures. It calls on PLANNER to satisfy goals and uses PLANNER theorems to hold the substantive knowledge (such as facts about integers) which are needed to prove properties of procedures.

INTENDER has three main uses for PLANNER:

1: It enables PLANNER to verify that its procedures do what is intended.

2: Most knowledge in PLANNER is embedded in procedures. INTENDER helps PLANNER understand these procedures and thus to have some knowledge of its own problem solving behaviour.

3: INTENDER enables PLANNER to verify that its plans (procedures) are valid relative to its procedural model of the world.

We shall express the properties of an expression x by the following function.

<INTENT

[-declarations-] |predecessor| |x| |function| -successors-> is true if |predecessor| evaluates to true, the function applied to the value of |x| is true, and the -successors- all evaluate to true. The value of the function intent is the value of |x|. The function intent is used to state a model for an expression x. As might be expected the models are stated in PLANNER. The intentions are established by INTENDER which is the language in which intentions are stated. The proof is by induction on the activations of the procedure. Thus for the control structure of LISP, the proof is by recursion induction. To avoid confusion we shall write the intention variables in upper case. Also we shall use !' to suppress invocations. Thus <+ 2 3> evaluates to the number 5 while !'<+ 2 3> evaluates to <+ 2 3>. For example the intentions in the prog below are all true.

```
<prog foo [[a 1] [b 2]]
        !;<intent [ ] <goal !'<= 1 .a>>>
        ;"Yes the identifier a was
                indeed initialized to 1. Will wonders never cease?"
        !;<intent [ ] <goal !'<= .b !'<+ .a 1>>>>
        !;<intent [ ]
                <goal !'<= .b 2>>
                <_ :b <+ .b 1>>
                <function [X ] <goal !'<= .X 3>>>
                <goal !'<=.b 3>>>
        ;"We have just verified that an assignment statement
                can change the value of the identifier b from 2 to 3"
        <.foo .b>
        ;"exit .foo with .b">
```

The following protocol for INTENDER verifies that the intentions in the above program do in fact hold. We shall use the notation |identifier|_|n| for the |n|th value of |identifier| and |idenifier|_ for the initial value.

```
        <assert !'<= 1 a_>>
        <assert !'<= 2 b_>>
        <goal !'<= 1 a_>>
        <goal !'<= b_ !'<+ a_ 1>>>
        <goal !'<= 2 b_>>
        <assert !'<= b_1 <+ b_ 1>>>
        <goal !'<= <+ b_ 1> 3>>
        <goal !'<= b_1 3>>
```

The essential idea for intentions comes from the break function introduced into LISP by W. Martin. An intention is not allowed to assign a value to a non-intention identifier and ordinary code is not allowed to reference intention identifiers. We shall distinguish intention identifiers from ordinary identifiers by putting them in all caps. The intention

```
        <INTEND
                [-declarations-] |predecessor| |expression;
```

|function|> is exactly like the function intent except that intention

variables can be declared in the declaration. In addition we need a function

        <OVERALL

                [-declarations-] |predecessor| |expression|

ction|> which is exactly like the function INTEND except that it is used to state the overall intention of a procedure. If |expression| is a junction then the overall input output intentions of the junction are given by |predecessor| and |function|. Thus INTENDER does computational induction across process boundaries. All the intentions in the function fact are true where

```
<define fact <function fact [n]
<overall [ ]
    <intention [ ]
        <goal !'<is? !'<non-neg> .n>>
        <assert  !'<is? !'<non-neg> .n>>>
    <repeat [[temp 1] [i 0]]
        !;<intention [ ]
            <prog [ ]
                <goal !'<is? !'<non-neg> .i>>
                <goal !'<= .temp !'<factorial .i>>>>
            <prog [ ]
                <assert !'<is? !'<non-neg> .i>>
                <assert !'<= .temp <factorial .i>>>>>
        <cond
          [<is? .n .i>
                <.fact .temp>
                ;"exit .fact with .temp"]>
        <_ :i <+ .i 1>>
        <_ :temp <* .i .temp>>>
     <function [X]
        <intention [ ]
                <assert !'<= .X !'<factorial .n>>>
                <goal !'<= .X !'<factorial .n>>>>>>>>
```

where

```
<define factorial <function  [n]
<overall [ ]
    <intention [ ]
```

```
          <goal !'<is? !'<non-neg> .n>>
          <assert !'<is? !'<non-neg> .n>>>
     <cond
          [<is? 0 .n>
               1]
          [¬"else"
               <* .n <factorial <- .n 1>>>]>
     <function [X]
          <intention [ ]
               <prog [ ]
                    <cond
                         [&<goal !'<= .n 0>>
                              <assert !'<= .X 1>>]
                         [<goal !'<not !'<= .n 0>>>
                              <assert
                                 !'<=
                                        .X
                                        !'<* .n !'<fact !'<- .n 1>>>>>]>
                    <assert !'<= .X !'<combinations .n 0>>>
                    <assert !'<=  .X !'<fact .n>>>>
               <prog [ ]
                    <cond
                         [&<goal !'<= .n 0>>
                              <goal !'<= .X 1>>]
                         [&<goal !'<not !'<= .n 0>>>
                              <goal !'<= .X !'<* .n !'<fact !'<- .n 1>>>>>]>
                    <goal !'<= .X !'<combinations .n 0>>>
                    <goal !'<=  .X !'<fact .n>>>>>>>>>
```

The following is a protocol of the action of INTENDER on the intentions of fact:

```
<assert !'<is? !'<non-neg> n_>>
enter intentions of repeat

Case 1:  initial entry
<assert !'<= 1 temp_>>
<assert !'<= 0 i_>>
     <goal !'<is? !'<non-neg> i_>>
     <goal !'<= 1 !'<factorial 0>>>
          enter intentions of factorial
                   n becomes 0
                   X becomes 1
                   <goal !'<is? !'<non-neg> 1>>
                   <goal !'<= 1 1>>
                   <assert !'<= 1 !'<factorial 0>>>
```

```
Case 2:  inductively assume
<assert !'<is? !'<non-neg> i_>>
<assert !'<= temp_ !'<factorial i_>>>
enter conditional

     Case 1:
     <assert !'<= n_ i_>>
          <goal !'<= temp_ !'<factorial n_>>>

     Case2:
     <assert !'<not !'<= n_ i_>>>
          <assert !'<= i_1 !'<+ i_ 1>>>
          <assert !'<=
                              temp_1
                              !'<* i_1 temp_>>>
          <goal !'<= temp_1 !'<factorial i_1>>>
          enter intentions of factorial
                    n becomes i_1
                    X becomes temp_1
                    <goal !'<is? !'<non-neg> i_1>>
                    <gcal !'<= 0 i_1>>
                    FAIL
                    <gcal !'<=
                         !'<*
                              i_1
                              !'<factorial !'<- i_1 1>>>
                         temp_1>>
```

On the other hand if INTENDER analyzes the intentions of factorial we get:

```
<assert !'<is? !'<non-neg> n_>>
enter conditional

     Case1:
     <assert !'<= 0 n_>>
          <goal !'<= 1 !'<fact 0>>>
          enter intentions of fact
                    n becomes 0
                    X becomes 1
                    <goal !'<= 0 0>>
                    <goal !'<= 1 1>>

     Case2:
     <assert !'<not !'<= 0 n_>>>
          <assert !'<=
                    !'<factorial !'<- n_ 1>>
                    !'<fact !'<- n_ 1>>>>
```

```
                <goal !'<=
                        !'<*
                                n_
                                !'<factorial !'<- n_ 1>>
                        !'<fact n_>>>>
                enter intentions of fact
                        n becomes n_
                        X becomes
                                !'<*
                                        n_
                                        !'<factorial !'<- n_ 1>>>
                        <goal !'<=
                                !'<*
                                        n_
                                        !'<factorial !'<- n_ 1>>>
                                !'<combinations n_ 0>>>
                        <goal !'<=
                                !'<*
                                        n_
                                        !'<factorial !'<- n_ 1>>>
                                !'<*
                                        n_
                                        !'<fact !'<- n_ 1>>>>>
```

The intentions for the function fctrl defined below are not so easy to establish.

```
<define fctrl <function fctrl [n]
<overall [[ARG .n]]
    intention [ ]
        <goal !'<is? !'<non-neg> .n>>
        <assert !'<is? !'<non-neg> .n>>>
   <repeat [[temp 1]]
        !;<intention [ ]
                <goal !'<= .temp !'<combinations .ARG .n>>>
                <assert !'<= .temp !'<combinations .ARG .n>>>>
        <cond
            [<is? 0 .n>
                <.fctrl .temp>
                ;"exit .fctrl with .temp"]>
        <_ :temp <* .temp .n>>
        <_ :n <- .n 1>>>
    <function [X]
        <intention [ ]
                <assert !'<= .X !'<factorial .ARG>>>
                <goal !'<= .X !'<factorial .ARG>>>>>>>>
```

We need to define an auxillary functicn in order to do the proof:

```
<define combinations <functicn [n r]
<overall [ ]
    <intention
        <and
                &<goal !'<is? !'<non-neg> .n>>
                &<goal !'<is? !'<non-neg> .r>>
                &<goal !'<is? !'<greater= .r> .n>>>
        <and
                &<assert !'<is? !'<non-neg> .n>>
                &<assert !'<is? !'<non-neg> .r>>
                &<assert !'<is? !'<greater= .r> .n>>>>
    <cond
        [<is? .n .r>
            1]
        [¬"else"
            <* .n <combinations <- .n 1> .r>>]>
    <function [X]
            <intention [ ]
                <prog [ ]
                        <cond
                            [&<goal !'<= .n .r>>
                                <assert !'<= 1 .X>>]
                            [&<goal !'<=  .r 0>>
                                <assert !'<= .X !'<factorial .n>>> ]>
                        <assert
                            !'<=
                                .X
                                !'<*
                                        !'<combinations !'<- .n 1> .r>
                                        .n>>>>
                <prog [ ]
                        <cond
                            [&<goal !'<= .n .r>>
                                <goal !'<= 1 .X>>]
                            [<goal !'<= .r 0>>
                        <goal !'<= .X !'<factorial .n>>> ]>
                        <goal
                            !'<=
                                .X
                                !'<*
                                        !'<combinations !'<- .n 1> .r>
                                        .n>>>>>>>>>>
```

INTENDER yields the following protocol for the intentions of

fctrl:

```
<assert !'<is? !'<non-neg> n_>>
enter intentions of repeat

Case 1: initial entry
    <assert !'<= 1 temp_>>
        <goal !'<= 1 !'<combinations n_ n_>>>
        enter intentions of combinations
                n becomes n_
                r becomes n_
                <goal !'<is? !'<non-neg> n_>>
                <goal !'<is? !'<non-neg> n_>>
                <goal !'<is? !'<greater= n_> n_>>
                <goal !'<= n_ n_>>
                <goal !'<= 1 1>>

Case 2:  inductively assume
    <assert
        !'<=
                temp_
                !'<combinations n_ n_>>>
enter conditional

    Case1:
    <assert !'<= 0 n_1>>
        <goal !'<= temp_ !'<factorial n_>>>
        enter intentions of factorial
                n becomes n_
                X becomes temp_
                <goal
                        !'<=
                                temp_
                                !'<combinations n_ 0>>>

    Case2:
    <assert !'<not !'<= 0 n_1>>>
        <assert
                !'<=
                        temp_1
                        !'<* temp_ n_1>>>
        <assert !'<= n_2 !'<- n_1 1> >>
        <goal !'<=
                temp_1
                !'<combinations n_ n_2>>>
        enter intentions of combinations
                n becomes n_
                r becomes n_2
                X becomes temp_1
                <goal !'<is? !'<non-neg> n_>>
                <goal !'<is? !'<non-neg> n_2>>
```

```
                              <goal !'<is? !'<greater= n_2> n_>>
                              <gcal !'<=
                                      temp_1
                                      !'<*
                                              !'<ccmbinations
                                                      n_
                                                      n_1>
                                              n_1>>>
```

We can use the same techniques for showing that a procedure will converge if its arguments satisfy certain conditions. The idea is tc define a partial order with no infinite descending chains and then prove that every time control goes through the same point in the program that it has descended in the partial order. The ordering we shall use is that [SMALLER [|a| |b| |c|] [|d| |e| |f|]] is true if one of the following three conditions holds:

|a| is less than |d|

|a|=|e| and |b| is less than |e|

|as|=|e| and |b|=|e| and |c| is less than 'f| For example consider Ackerman's functicn as defined below:

```
<define ackerman <function [z x y.
<overall [ ]
    <intention [ ]
        <prog [ ]
                <goal !'<is? !'<non-neg> .z>>
                <goal !'<is? !'<non-neg> .x>>
                <goal !'<is? !'<non-neg> .y>>>
        <prog [ ]
                <assert !'<is? !'<non-neg> .z>>
                <assert !'<is? !'<non-neg> .x>>
                <assert !'<is? !'<non-neg> .y>>>
        smaller>
    <cond
          [<is? 0 .x>
                <rule [ ] .z
                        [0 .y]
                        [1 0]
                        [<greater 1> 1]>]
```

```
              [<is? 0 .z>
                  <+
                          <ackerman 0 <- .x 1> .y>
                          1>]
              [¬"else"
                  <ackerman
                          <- .z 1>
                          <ackerman .z <- .x 1> .y>
                          .y>]>
      <function [w]
          <intention [ ]
                  <assert !'<is? !'<non-neg> .w>>
                  <goal !'<is? !'<non-neg> .w>>>>>>>
```

```
<define show-smaller <consequent [a b c d e f]
        [smaller [?a ?b ?c] [?a ?e ?f]]
        <cond
            [&<goal !'<is? !'<less ?d> ?a>>]
            [&<goal !'<= ?a ?d>>
                <cond
                    [&<goal !'<is? !'<less ?e> ?b>>]
                    [&<goal !'<= ?b ?e>>
                        <goal !'<is? !'<less ?f> ?c>>]
                    [¬"else" <fail>]>]
            [¬"else" <fail>]>>>
```

The protocol for PLANNER on ackerman's function is:

```
        <assert !'<is? !'<non-neg> z_>>
        <assert !'<is? !'<non-neg> x_>>
        <assert !'<is? !'<non-neg> y_>>
        enter conditional

            Case 1:
            <assert !  = 0 x_>>

        <assert !'<is? !'<greater 0> x_>>

            Case 2:
            <assert !'<= 0 z_>>
                <goal [smaller
                        [0 !'<- x_ 1> y_]
                        [0 x_ y_]]>
                enter show-smaller
                    a becomes 0
                    b becomes !'<- x_ 1>
                    c becomes y_
                    d becomes 0
                    e becomes x_
```

```
                                f becomes y_
                                     <goal !'<is !'<less 0> 0>>
                                     FAIL
                                     <goal !'<= 0 0>>
                                     <gcal !'<less x_> !'<- x_ 1>>

                <assert !'<is? !'<greater 0> z>>

                     Case 3:
                          <goal [smaller
                                    [z_ !'<- x_ 1> y_]
                                    [z_ x_ y_]]>
                          <goal [smaller
                                    [!'<- z_ 1> <ackerman z_ !'<- x_ 1> y_> y_>]
                                    [z_ x_ y_]]>
```

We would like to show that if we reverse a list twice then we get the original list.

```
<define reverse <function rev [l]
<overall [ ]
      t
     <repeat [[u .l] [v ()]]
         !;<intention [ ]
                  <goal !'<is? .v !'<reverse !'<sub .l .u>>>>
                  <assert !'<is? .v !'<reverse !'<sub .l .u>>>>>
         <cond
             [<empty? .u>
                  <.rev .v>
                  ;"exit .rev with .v"]>
         <_ :v (<1 .u> !.v)>
         <_ :u <rest .u>>>
     <function [X]
         <intention [ ]
                  <and
                         <cond
                             [<is? () .l>
                                  <assert !'<= .X ()>>]>
                         <assert !'<is? .X !'<rev .l>>>
                         <assert !'<is? .l !'<reverse .X>>>>
                  <prog [ ]
                         <cond
                             [<is? () .l>
                                  &<goal !'<= .X ()>>]>
                         &<goal !'<is? .X !'<rev .l>>>
                         &<goal !'<is? .l !'<reverse .X>>>>>>>>>>
```

We would like to show that for all |l| that <reverse <reverse |l|>> is |l|. Again we will need a helping function to express our intentions. We shall define <SUB |x| |y|> to be |x| subtract |y| as lists.

```
<define last <function [x]
        <cond
            [<empty? <rest .x>>
                <1 .x>]
            [¬"else"
                <last <rest .x>>]>>>

<define butlast <function [x]
        <cond
            [<empty? <rest .x>>
                () ]
            [¬"else"
                (<1 .x> <butlast <rest .x>>]>>>

<define sub <function [x y]
<overall [ ]
        t
        <cond
            [<is? .x .y>
                () ]
            [¬"else"
                (<1 .x> {sub <rest .x>} .y) ]>
        <function [Z]
            <intention [ ]
                <cond
                    [&<goal !'<is? .y ()>>
                        <assert !'<is? .Z .x>>]
                    [&<goal !'<not !'<is? .y ()>>>
                        <assert !'<=
                                !'<last !'<sub .x !'<rest .y>>>
                                !'<1 .y>>>
                        <assert !'<=
                                .Z
                                !'<butlast !'<sub .x !'<rest .y>>>>> ]>
                <cond
                    [&<goal !'<is? .y ()>>
                        <goal !'<is? .Z .x>>]
                    [&<goal !'<not !'<is? .y ()>>>
                        <goal !'<=
                                !'<last !'<sub .x !'<rest .y>>>
                                !'<1 .y>>>
                        <goal !'<=
                                .Z
                                !'<butlast !'<sub .x !'<rest
.y>>>>>]>>>>>>
```

```
<define rev <function [list]
<overall [ ]
        t
        <cond
            [<monad? .list>
                .list ]
            [¬"else"
                (<last .list> !<rev <butlast .list>>) ]>
        <function [X]
            <intention [ ]
                <prog [ ]
                        <assert !'<is? .X !'<reverse .list>>>
                        <assert !'<is? .list !'<reverse .X>>>>
                <prog [ ]
                        <goal !'<is? .X !'<reverse .list>>>
                        <goal !'<is? .list !'<reverse .X>>>>>>>>>
```

```
The protocol of INTENDER on REVERSE is:
        enter intenticns of repeat

        Case 1:  initial entry
            <assert !'<= u_ l_>>
            <assert !'<= v_ ()>>
                <goal !'<is? () !'<reverse !'<sub l_ l_>>>>
                enter intentions of sub
                        x becomes ()
                        y beccmes ()
                        <assert !'<= () !'<sub l_ l_>>>
                enter intentions of reverse
                        l becomes ()
                        <assert !'<= () !'<reverse ()>>>

        Case 2:  inductive hypothesis
        <assert !'<is v_ !'<reverse !'<sub l_ u_>>>>
            enter conditional

                Case 1:
                <assert !'<= () u_>>
                        enter overall consequent
                            X becomes v_
                                <goal !'<is v_ !'<reverse l_>>>

            <assert !'<not !'<= () u_>>>

                Case 2:
                        <assert !'<= v_1 (!'<1 u_> !'(value v})>>
                        <assert !'<= u_1 !'<rest u_>>>
                        <goal !'<is
                                      (
                                          !'<1 u_>
```

```
                                  !'(reverse !'<sub l_ u_>})
                              !'<reverse
                                  !'<sub
                                         l_
                                         !'<rest u_>>>>>
```

Allowing shared side effects in structured data considerably complicates the process of proving intentions.

7.1.2 Models in Patterns: Aims

Aims are like intentions except that they are actors and occur in patterns.

<AIM predecessor pattern down up successors> is the form for a call to the actor aim. An aim will be said to be attained when the following conditions are satisfied:

[1] Its predecessor evaluates to true

[2] We apply the function down with two arguments. The first is the expression to be matched. The second is <> if and only if pattern doesn't match.

[3] We apply the function up with two arguments. The first is <> if and only if the rest of the pattern doesn't match. The second is <> if and only if pattern fails.

[4] The successors evaluate to true.

The function down expresses the intent of the downward action of the pattern and the function up expresses the upward going action. The actor <AIMING [declarations] predecessor pattern down up successors> is exactly like the actor AIM except that intention variables may be declared. For example the aim in the folowing expression is attained:

```
<aiming [[OLD-F .f]]
        t
        _f
        <function [X Y]
                <assert !'<eq .f .X>>
                <assert !'<is? .Y t>>>
        <function [X Y]
                <cond
                    [&<goal !'<is?  .X <>>>
                        <assert !'<eq .f .OLD-F>>
                        <assert !'<is? .Y [ ]>>]
                    [&<goal !'<is? .X t>>
                        <assert !'<eq .f .X>>
                        <assert !'<is? .Y t>>]>>>
```

The value of f changes only if the rest of the match succeeds. The actor <ENTIRE [declarations] predecessor pattern down up successors> is exactly like the actor AIMING except that it is used to express the entire intent of the pattern. For example for the actor ATOMIC which takes no arguments and matches only atoms can be characterized by:

```
<define atomic <actor [ ]
<entire [ ]
        t
        <atomic>
        <function [X Y]
                <cond
                   [&<goal !'<atom .X>>
                        <assert !'<is? .Y t>>]
                    [&<goal !'<not !'<atom .X>>>
                        <assert !'<is? .Y <>>>]>>
        <function [X Y]
                <assert !'<is? .X .Y>>>>>>
```

7.1.3 Models of PLANNER Theorems

We shall construct models for PLANNER theorems in much the same manner as for MATCHLESS patterns.

<THINTENT predecessor x down up successors> is true if the following conditions are met:

[1] the predecessor is true.

[2] We apply the function down with two arguments: The first argument is <> if and only if the evaluation of x fails. If the first argument is not <> then the value of the second argument is the value of x.

[3] We apply the function up with four arguments. The first is <> if and only if the rest of the computation fails. If the first argument is <> then the second argument is the message of the failure. The third argument is <> if and only if the evaluation of x fails. If the third argument is not <> then the fourth argument is the value of x.

The function THINTEND is exactly like the function THINTENT except that a declaration of intention variables must be the first argument. For example the folliwng intention is always satisfied: Recall that the function ASSERT1 will assert a statement if has not already been proved.

```
<thintend [[already-proved <>]]
          t
          <assert1 [subset a b]>
          <function [X Y]
                    <cond
                         [&<goal [proved [subset a b]]>
                              <assert !'<is? .X <>>>
                              <_ :already-proved t>
```

```
                <assert !'<is? .Y <>>> ]
           [&<goal !'<not [proved [subset a b]]>>
                <assert [proved [subset a b]]>
                <assert !'<is? .X t>>
                <assert !'<is? .Y [subset a b]>>]>>
<function [X Y U V]
        <cond
           [<is? <> .already-proved>
                <cond
                    [&<goal !'<is? .X <>>>
                        <erase [proved [subset a b]]>]>]>
        <assert !'<is? .U .X>>
        <assert !'<is? .V .Y>>>>
```

7.2 Teaching Procedures

Crucial to our understanding of the phenomenon of teaching is the teaching of procedures. Understanding the teaching of procedures is crucial because of the central role played by the structrual analysis of procedures in the foundations of problem solving. How can procedures such as multiplication, algebraic simplification, and verbal analogy problem solving be taught efficiently? Once these procedures have been taught, how can most effective use of them be made to teach other procedures? In addition to being incorporated directly as a black box, a procedure which has already been taught can be used as a model for teaching other procedures with an analogous structure. One of the most important methods of teaching procedures is telling. For example one can be told the algorithm for doing symbolic integration. Telling should be done in a high level goal-oriented language. PLANNER goes a certain distance toward raising the level of the language in which we can express a procedure to a computer. The language has primitives which implement fundamental problem solving abilities. Teaching procedures is intimately tied to what superficially appears to be the special case of teaching procedures which write procedures. The process of teaching a procedure should not be confused with the process of trying to get the one being taught to guess what some black box procedure really does [as is the case in in sequence extrapolation for example]. The teacher is duty bound to tell anything that might help the one being taught to understand the properties and structure of the procedure. We assume that the teacher has a good model of how the student thinks. Also, just because we speak of "teaching", we do not thereby assume that anything like what classically has been called learning is taking place in the student. However, this does not exclude the possiblity that the easiest way to teach many procedures is through examples. We can give protocols of the action of the procedure for various inputs and enviroments. By "variablization" [the introduction of identifiers for the constants of the examples] the protocols can be formed into a tree. Then a recursive procedure can be generated by identifying indistinguishable nodes on the tree. We call the above procedure for constructing procedures from examples the procedural abstraction of protocols. ..ocedural abstraction can be used to teach oneself a procedure.

7.2.2 By Procedural Abstraction

7.2.2.4 Examples of Procedural Abstraction

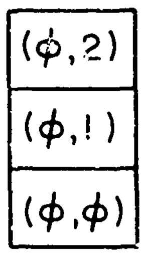
7.2.2.4.1 Building a Wall

We shall explain procedural abstraction in more detail using the example of building a wall. We define <brick-at |w| |h|> to mean that there is a brick at the location with width |w| and height |h|

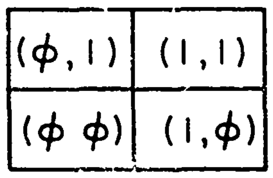
7.2 page

310 A

BUILDING WALLS



[WALL 1 2]



[WALL 2 1]

NOTE: THE NUMBERS IN THE BOXES REPRESENT THE COORDINATES OF THE BRICKS.

and define the statement [wall |w| |h|] to mean that there is a wall of width w and height h using the definition

```
<define wall <function [w h]
        <conjunction [[ww 0]]
            [¬"inc" ww ¬"by" 1 ¬"thru" .w]
                <conjunction [[hh .h]]
                    [¬"dec" hh ¬"by" 1 ¬"thru" 0]
                        <brick-at .ww .hh>>>>>
```

Thus <wall 1 2> means

```
        <and
                <and
                        <brick-at 0 2>
                        <brick-at 0 1>
                        <brick-at 0 0>>
                <and
                        <brick-at 1 2>
                        <brick-at 1 1>
                        <brick-at 1 0>>>.
```

Notice that the syntactic definition of a wall runs orthogonal to the way in which a wall has to be constructed. Thus we could not use purely syntax directed methods to construct walls. The predicate <ASSIGNED? var> is true only if the identifier var has been assigned a value.

```
<define build-tower
 <consequent build
        [[!=fir w h] [actions [ ]]]
            [brick-at ?w ?h]
                <cond
                     [<not? <assigned? h>>
                         <_ .h 0>]>
                <cond
                    [<current? [brick-at ?w ?h]>
                        <.build ()>
                        ;"exit .build with () ]>
                    [<is? .h 0>
                        <.build (!'<put-brick-at ?w ?h>)>]>
                <_ :actions <goal [brick-at ?w <- .h 1>]>>
```

```
                  <goal [put-brick-at ?w ?h]>
                  <goal [check-brick-at .w .h]>
                  <assert [brick-at .w .h]>
                  <.build (!.actions !'<put-brick-at .w .h>)>>
```

If we give PLANNER the task of constructing a [wall 1 2], then the actions that will be taken are:

```
        <put-brick-at 0 0>
        <put-brick-at 0 1>
        <put-brick-at 0 2>
```

If the goal is [wall 2 1] then the actions are:

```
        <put-brick-at 0 0>
        <put-brick-at 0 1>
        <put-brick-at 1 0>
        <put-brick-at 1 1>
```

We shall use the expression new 5 to mean that a new identifier is bound and initialized to 5. We shall use the expression <value 9> to mean a reference to an identifier whose value is 9; the expression <alter 3 7> means that an identifier with value 3 is altered to be the value 7. More precisely, the protocol for [wall 1 2] is

```
<new [1 2]
<new [UNASSIGNED UNASSIGNED]
<_ <alter UNASSIGNED 0> 0>
<is? <value 0> <value 1>> IS FALSE
SO
<_ <alter UNASSIGNED 0> 0>
<is? <value 0> <value 2>> IS FALSE
    SO
        <put-brick-at <value 0> <value 0>>
        <_ <alter 0 1> <+ <value 0> 1>>
        <is? <value 1> <value 2>> IS FALSE
            SO
                <put-brick-at <value 0> <value 1>>
                <_ <alter 1 2> <+ <value 1> 1>>
                <is? <value 2> <value 2>> IS TRUE
                    SO
                        <_ <alter 0 1> <+ <value 0> 1>>
                        <is? <value 1> <value 1>> IS TRUE
                            SO
                                [ ]>>
```

The protocol for [wall 2 1] is

```
<new [2 1]
<new [UNASSIGNED  UNASSIGNED]
<_ <alter UNASSIGNED 0> 0>
<is? <value 0> <value 2>> IS FALSE
 SO
<_ <alter UNASSIGNED 0> 0> <is? <value 0> <value 1>> IS FALSE
     SO
         <put-brick-at <value 0> <value 0>>
         <_ <alter 0 1> <+ <value 0> 1>>
         <is? <value 1> <value 1>> IS TRUE
               SO
                  <_ <alter 0 1> <+ <value 0> 1>>
                  <is? <value 1> <value 2>> IS FALSE
                        SO
                           <_ <alter 1 0> 0>
                           <is? <value 0> 1>IS FALSE
                                 SO
                                    <put-brick-at <value 1> <value 0>>
                                    <_ <alter 0 1> <+ <value 0> 1>>
                                    <is? <value 1> <value 1>> IS TRUE
                                          SO
                                             <_
                                                 <alter 1 2>
                                                 <+ <value 1> 1>>
                                             <is?
                                                 <value 2>
                                                 <value 2>> IS TRUE
                                                      SO [ ]>>
```

The protocol for [wall 2 2] is

```
<new [2 1]
         <new [UNASSIGNED UNASSIGNED]
         <_ <alter UNASSIGNED 0> 0>
         <is? <value 0> <value 2>> IS FALSE
         SO
         <_ <alter UNASSIGNED 0> 0>
         <is? <value 0> <value 2>> IS FALSE
         SO
         <put-brick-at <value 0> <value 0>>
         <_ <alter 0 1> <+ <value 0> 1>>
         <is? <value 1> <value 2>> IS FALSE
         SO
         <put-brick-at <value 0> <value 1>>
         <_ <alter 1 2> <+ <value 1> 1>>
         <is? <value 2> <value 2>> IS TRUE
         SO
         <_ <alter 0 1> <+ <value 0> 1>>
         <is? <value 1> <value 2>> IS FALSE
```

```
          SO
          <_ <alter 2 0> 0>
          <is? <value 0> <value 2>> IS FALSE
          SO
          <put-brick-at 1 0>
          <_ <alter 0 1> <+ <value 0> 1>>
          <is? <value 1> <value 2>> IS FALSE
          SO
          <put-brick-at <value 1> <value 1>>
          <_ <alter 1 2> <+ <value 1> 1>>
          <_ <alter 1 2> <+ <value 1> 1>>
          <is? <value 2> <value 2>> IS TRUE
          SO
          [ ]>>
```

By introducing identifiers for the constants and by tracing the bindings of the identifiers of BUILD-TOWER the protocols can be arranged in a tree as follows:

```
new [w h]
new [ww=UNASSIGNED; hh=UNASSIGNED]
<_ :ww 0>
if <is? .ww .w>
then
[ ]
 else <_ :hh 0> if <is? .hh .h>
      then
         <_ :ww <+ .ww 1>>
         if <is? .ww .w>
              then
                 [ ]
              else
                 <_ :hh 0>
                 if <is? .hh 0>
                      then
                         <_ :ww <+ .ww 1>>
                                 if <is? .ww .w>
                                      then
                                         [ ]
                                      else...
                      else...
      else
         <put-brick-at .ww .hh>
         <_ :hh <+ .hh 1>>
         if <is? .hh .h>
              then
                 <_ :ww <+ .ww 1>>
                 if <is? .ww 1>
                      then
                         [ ]
```

```
                    else
                       <_ :hh 0>
                       if <is? .hh .h>
                            then
                               <put-brick-at .ww .hh>
                               <_ :hh <+ .hh 1>>
                                        if <is? .hh .h>
                                             then
                                                <_ :ww <+ .ww 1>>
                                                if <is? .ww .w>
                                                     then [ ]
                                                     else...
                                            else...
                            else...
            else
               <_ :hh <+ .hh 1>>
               if <is? .hh .h>
                    then
                       <_ :ww <+ .ww 1>>
                       if <is? .ww .w>
                            then [ ]
                            else...
                    else...
```

We define the protocol of an evaluation to be a list of the events and the places in the program where they happen that occur when the evaluation is being carried out. By examining the protocols of the system as it tries to build a wall we find that it always uses the same procedure. Of course it will not always be the case that the protocols from the solutions of the instances of a goal can be combined into a procedure. The basic idea is to combine the set of protocols into a tree and then consider any two nodes of the tree which cannot be distinguished on the basis of the protocols to be identical. In other words it is necessary to compute a minimal or almost minimal homomorphic image of the set of available protocols. Unfortunately it is often difficult to extract the information needed to do procedural abstraction from the protocols produced by PLANNER theorems as they solve problems. The procedure that the theorem is in fact using can be expressed as follows:

```
<define compile-build <function [w h]
<overall [ ]
    !;<intention [ ]
        <and
                <goal !'<is? !'<non-neg> .w>>
                <goal !'<is? !'<non-neg> .h>>>>
        <and
                <assert !'<is? !'<non-neg> .w>>
                <assert !'<is? !'<non-neg> .h>>>>
    <repeat column
        [[ww 0]]
        !;<intention [ ]
                <goal [wall .ww .h]>
                <assert [wall .ww .h]>>
        <cond
            [<is? .ww .w>
                <intent <wall .w .h>>
                <.column>
                ;"exit .column"]>
        <repeat height [[hh 0]]
                !;<intention [ ]
                        <goal [column .ww .hh]>
                        <assert [column .ww .hh]>>
                <cond
                    [<is? .hh .h>
                        <.height>
                        ;"exit .height with <>"]>
                !;<intent <goal [support-for .ww .hh]>>
                <put-brick-at .ww .hh>
                !;<intent <goal [brick-at .ww .hh]>>
                <_ :hh <+ .hh 1>>>
        <_ :ww <+ .ww 1>>>
      <function [X]
        <assert [wall .w .h]>
        <goal [wall .w .h]>>>>

<define  check-wall
     <consequent
        check-wall
        [w' w h' h]
        [wall ?w' ?h']
        <cond
                [<or?
```

```
                    &<goal !'<is? ?h' 0>>
                    &<goal !'<is? ?w' 0>>>]
               [<is? !'<+ ?h 1> .h'>
                    <goal [wall ?w' .h]>
                    <goal [column ?w' ?h']>]
               [<is? !'<+ ?w 1> .w'>
                    <goal [wall ?w .h']>
                    <goal [column ?w' ?h']>]
               [¬"else"
                    <fail <> .check-wall>]>>>

<define check-column
   <consequent
        check-column
        [w h h']
        [column ?w ?h']
        <cond
           [&<goal !'<is? ?h' 0>>]
           [<is? !'<+ ?h 1> .h'>
                <goal [column ?w ?h]>]
           [¬"else"
                <fail <> .check-column>]>>>

<define check-support
    <consequent
        check-support
        [w h]
        [support-for ?w ?h]
        <cond
           [&<goal !'<is? ?h 0>>]
            [<goal [column .ww .hh]>]
           [¬"else"
                <fail <> .check-support>]>>>

<define put-brick-at
    <function [w h]
        <overall [ ]
                <goal [support-for .w .h]>
                <put-brick-at .w .h>
                <assert [brick-at .w .h]>>>>
```

The INTENDER protocol for the verification of the intentions of compile-build is:

```
        <assert !'<is? !'<non-neg> w_>>
        <assert !'<is? !'<non-neg> h_>>
        enter intentions of repeat
```

```
Case 1: initial entry
<assert !'<is? 0 ww_>>
    <goal [wall 0 h_]>
        enter intentions of check-wall
                w' becomes 0
                h' becomes h_
                <goal !'<is? h_ 0>>
                FAIL
                <goal !'<is? 0 0>>

Case 2: Inductively assume
<assert [wall ww_ h_]>
enter conditional

    Case 1:
    <assert !'<is? w_ ww_>>
        <goal [wall w_ h_]>

    Case 2:
    <assert !'<not !'<is? w_ ww_>>>
        enter intentions of repeat

        Case 1:  initial entry
        <assert !'<is? 0 hh_>>
            <goal [column ww_ hh_]>
                enter intentions of check-column
                w becomes ww_
                h' becomes hh_
                <goal !'<is? 0 hh_>>

        Case 2: inductively assume
        <assert [column ww_ hh_]>
        enter conditional

            Case 1:
            <assert !'<is? hh_ h_>>
                <assert !'<is? ww_1 !'<+ ww_ 1>>>
                <goal [wall ww_1 h_]>
                enter intentions of check-wall
                        w' becomes ww_1
                        h' becomes h_
                        <goal !'<is? !'<+ ?h 1> h_1>>
                        w becomes w_
                        <goal [wall ww_ h_]>

            Case 2:
            <assert !'<not !'<is? hh_ h_>>>
                <goal [support-for ww_ hh_]>
                enter intentions of check-support
                        w becomes ww_
                h becomes hh_
                <goal [column ww_ hh_]>
        <assert !'<is? hh_1 !'<+ hh_ 1>>>
        <goal [column ww_ hh_1]>
        enter intentions of check-column
                w becomes ww_
                h' becomes hh_1
                <goal !'<is? !'<+ ?h 1> hh_1>>
                h becomes hh_1
                <goal [column ww_ hh_1]>
```

Note that the above proof that COMPILE-BUILD meets its intentions is relative to the PROCEDURAL MODEL that we have constructed. The procedural model is constructed out of procedures such as PUT-BRICK-AT. The procedural model is connected to our goal oriented language by CORRESPONDENCE RULES such as CHECK-SUPPORT.

The structure of the abstracted procedure must at least reflect the structure of the PLANNER theorems from which it has been abstracted. Thus the abstraction of a for-proved loop will generate a recursive equation which might be simplified to a loop. Some of the recursion in abstracted functions is primarily generated by the structure of the data of the problem. If we consider the tags column and height to define functions, then the proof is essentially by recursion induction. In the above procedure .w is the width of the wall to be built, .ww is a running index over the width, .h is the height, and .hh is a running index over the height. Using the intentions in the above procedure as subgoals we can easily see that the procedure does build walls. Notice that we can use the protocols of the procedure [in a process that we call "protocol rejection"] to reject false subgoals in much the same way that Gelernter used diagrams in his geometry theorem prover. For example we might evaluate <compile-build 1 2>, <compile-build 2 1>, and <compile-build 3 2> remembering the protocols of the evaluations. Thus when considering the case where the intention

```
                <intent
                    <or
                        <is? .ww 0>
                        <wall <sub1 .ww> .hh>>>
```

is evaluated immediately after <end column> is evaluated, it will be the case that <is? .ww 0> is false and so cannot possibly be a provable subgoal even though it implies the intention. The subgoal will be to prove [implies <not <is? .w 0>> <wall <sub1 .ww> .hh>]. Of course using protocols for the purpose of rejecting false subgoals does not help us to eliminate those that are true but unprovable.

7.2.2.4.2 Rever. ing a List at All Levels

Consider the following protocols for a procedure r:

```
<new [a]
<is? <monadic> <value a>>> IS TRUE
SO <value a>

thus <r a> is a

<new [[n]]
<is? <monadic> <value [n]>> IS FALSE
     SO
        [
         {new [<rest <value [n]>>]
         <is? <monadic> <value [ ]>> IS TRUE
              SO <value [ ]>}
```

```
          <new [<<value [n]> 1>]
          <is? <monadic> <value n>> IS TRUE
                SC <value n>>]>

thus <r [n]> is [n]


<new [[a b]]
<is? <monadic> <value [a b]>> IS FALSE
      SO
          [
          [new [<rest <value [a b]>>]
          <is? <monadic> <value [b]>> IS FALSE
                SO
                    [
                    [<new [<rest <value [b]>>]
                    <is? <monadic> <value [ ]>> IS TRUE
                          SC <value [ ]>>]
                    <new [<<value [b]> 1>]
                    <is? <monadic> <value b>> IS TRUE
                          SC <value b>>]]
          <new [<<value [a b]> 1>]
          <is? <monadic> <value a>> IS TRUE
                SC <value a>>]>

thus <r [a b]> is [b a]


<new [[[a]]]
<is? <monadic> <value [[a]]>> IS FALSE
      SO
          [
          [<new [<rest <value [[a]]>>]
          <is? <mcnadic> <value [ ]>> IS TRUE
                SO [ ]>]
          <new [<<value [[a]]> 1>]
          <is? <mcnadic> <value [a]>> IS FALSE
                SO
                    [
                    [<new [<rest <value [a]>>]
                    <is? <monadic> <value [ ]>> IS TRUE
                          SC [ ]>]
                    <new [<<value [a]> 1>]
                    <is? <monadic> <value a>> IS TRUE
                          SC <value a>>]>]>

thus <r [[a]]> is [[a]]
```

We obtain the following prctoccl tree:

```
<new [x1]
if <is? <monadic> .x1>
     then .x1
     else
        [
        {new [x2 <rest .x1>]
        if <is? <monadic> .x2>
             then .x2
             else
                [
                {new [x3 <rest .x2>]
                if <is? <monadic> .x3>
                     then .x3
                     else...}
                <new [x4 <1 .x2>]
                if <is? <monadic> .x4>
                     then .x4
                     else...>]}
        <new [x5 <1 .x1>]
        if <is? <monadic> .x5>
             then .x5
             else
                [
                {new [x6 <rest .x5>]
                if <is? <monadic> .x6>
                     then .x6
                     else...}
                <new [x7 <1 .x5>]
                if <is? <monadic> .x7>
                     then .x7
                     else...>]>]>
```

By identifying indistinguishable nodes we obtain:

```
<define super-reverse <function [x]
          <cond
               [<is? <monadic> .x>
                   .x]
               [¬"else"
                    [
                              {super-reverse <rest .x>}
                              <super-reverse <1 .x>>]]>>>
```

7.2.2.4.3 Finding the Description of a Stick

Suppose that we have the following data base:

[block a]

[block b]

[glued a b]

The above data base represents a stick on the the basis of the following protocol:

```
<goal [stick a b]>
        <new [UNASSIGNED UNASSIGNED UNASSIGNED]
                ;"we have three new identifiers that do not have
values"
        consequent: [stick <given UNASSIGNED a> <given UNASSIGNED b>]
        cond
                <current [glued <given a> <given b>]>
                <return t>>
```

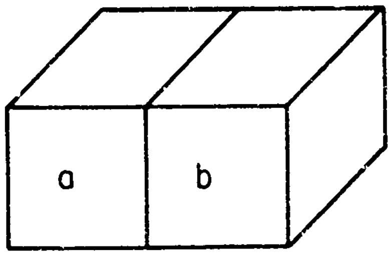
Now suppose that the data base is:

```
        [block a]
        [block b]
        [block c]
        [glued a b]
        [glued b c]
        [between a b c]
```

We obtain the following protocol:

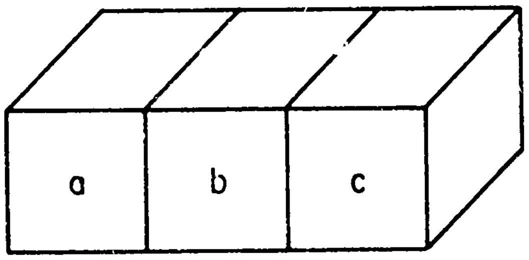
```
<goal [stick a c]>
        [new UNASSIGNED UNASSI[illegible]SIGNED]
        consequent:  [stick <g[illegible]iven c>]
        cond
                <current [glued <given a> <given c>]>
                fail
        <current [block <given a>]>
        <goal [glued <value a> <_ UNASSIGNED b>]>
        <current [between <value a> <value b> <given c>]>
        <goal [stick <value b> <value c>]>
                [new UNASSIGNED UNASSIGNED UNASSIGNED]
                consequent:  [stick <given b> <given c>]
                cond
                        <proved [glued <given b> <given c>]>
                        <return t>
```

7.2 page 324



[block a]
[block b]
[glued a b]

FIGURE 1



[block a]
[block b]
[block c]
[glued a b]
[glued b c]
[between a b c]

FIGURE 2

[block a]
[block b]
[block c]
[glued a b]
[glued b c]
[not[between a b c]]

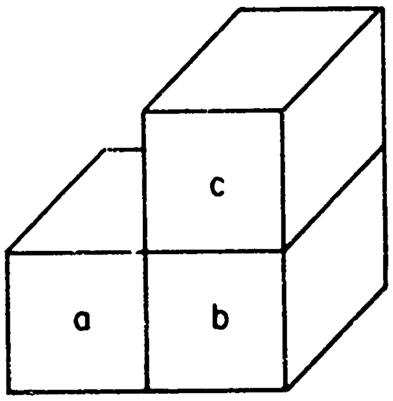


FIGURE 3

By variabalization we obtain the following protocol tree:

```
<goal [stick u v]>
       [new x y z]
       consequent c1: [stick ?x ?z]
       <cond
              [&<goal [glued ?x ?z]>
              <.c1 t>
              ;"exit .c1 with t"]>
       <current [block ?x]>
       <goal [glued .x _y]>
       <current [between .x .y ?z]>
       <goal [stick .y .z]>
              [new x1 y1 z1
              consequent c2:  [stick ?x1 ?z1]
              <cond
                     [&<goal [glued ?x1 ?z1]>
                     <.c2 t>
                     ;"exit .c2 with t"]>
              <current [block ?x1]>
              <goal [glued .x1 _y1]>
              <current [between .x1 .y1 ?z1]>
              <goal [stick .y1 .z1]>
```

By identifying indistinguishable nodes we obtain the following consequent theorem which is the description of a stick.

```
<define stick-des [illegible]tion <consequent c
       [x y z]
       [stick ?x ?z]
              <cond
                     [illegible]<goal [glued ?x ?z]>
                     [illegible] t>
                     [illegible] .c with t"]>
              <curren[illegible] ?x]>
              <goal [g[illegible] .x _y]>
              <current [between .x .y ?z]>
              <goal [sti[illegible] .y .z]>>>
```

7.2.2.4.4 Finding the Fibonocci Numbers Iteratively

Sometimes it is possible to improve the efficiency of a procedure by procedural abstraction. For example consider the protocols of the schema f defined below.

```
<define f <function [n]
          <cond
              [<or <P .n> <P <S .n>>>
                   <ONE>]
              [¬"else"
                   <A <f <S .n>> <f <S <S .n>>>>]>>>
```

We shall used the abbreviation that <f¬0 x> is x and <f¬n+1 x> is <f <f¬n x>> where f is a function. Thus <f¬2 x> is <f <f x>>. The protocol for the above schema is:

```
if <or <P <S¬0 n>> <P <S¬1 n>>>
      then <ONE>
      else
          <A
                    if <or <P <S¬1 n>> <P <S¬2 n>>>
                          then <ONE>
                          else
                              <A
                                        if <or <P <S¬2 n>> <P <S¬3 n>>>
                                              then <ONE>
                                              else...
                                        if <or <P <S¬3 n>> <P <S¬4 n>>>
                                              then <ONE>
                                              else...>
                    if <or <P <S¬2 n>> <P <S¬3 n>>>
                          then <ONE>
                          else
                              <A
                                        if <or <P <S¬3 n>> <P <S¬4 n>>>
                                              then <ONE>
                                              else...
                                        if <or <P <S¬4 n>>  <P <S¬5 n>>>
                                              then <ONE>
                                              else...>>
```

By procedural abstraction we can obtain a function f1 which is equivalent to f. The function is obtained by identifying some of the nodes that are not on the same branch of the protocol tree.

```
<define f <function out [n]
        <cond
             [<or <P .n> <P <S .n>>>
                 <.out <ONE> <ONE>>]
             [¬"else"
                 <call
                          <f <S .n>>
                          <function [down1 down2]
                                   <.out
                                           <A .down1 .down2>
                                           .down1>>>]>>>
```

Another approach is to use some of the theory of recursive schemas. The function f defined above is schematically equivalent to the function ff defined below

```
<define ff <function ff [n]
        <for [[x 0] [y 0]]
                          [[¬"test" <P .n> <.ff .x> ;"exit .ff with .x"]
                          [¬"step" <_ :n <S .n>>]]
        <_ [:x :y] <tuple <A .x .y> .x>>
                 ;"the previous statement is just a tricky way to
                          simultaneously accomplish <_ :x <A .x .y>>
                          and <_ :y .x>">>>
```

Note that <fib n> the nth Fibonacci number can be defined as follows

```
<define fib <function  [n]
        <cond
            [<or <is? 1 .n> <is? 2 .n>>
                 1]
            [¬"else"
                 <+ <fib <- .n 1>> <- .n 2>>]>>>
```

Using the interpretation that <ONE> is 1, <P x> tests to see if x is 1, and A is add, we see that the function fib can be rewritten iteratively.

The process of procedural abstraction is very much like a generalized form of compilation. The relationship between the compiled version and the interpreted version can be very subtle. In classical compilers the relationship is much more straightforward. Every time that the interpreter for the language changes the compiler must change. In fact the interpreter and compiler are two modes of what is essentially one program: an interpreter-compiler. In compile mode it would actually produce the compiled code for the source code; in interpret mode it would take the actions corresponding to the compiled code that would be produced in compile mode. The interpreter-compiler can be written in MATCHLESS so that in compile mode the MATCHLESS skeletons have as value the compiled code. One problem with interpreter-compilers is that they suffer from the inefficiency of double interpretation. Instead of directly interpeting the expressions, in interpret mode the interpeter-compiler interprets the skeletons that would produce the code in compile mode. The problem can be solved by compiling the interpreter-compiler for interpret mode. We would like to try to extend this idea to PLANNER in a more nontrivial way so that goals would be created to produce the compiled code.

7.2.2.4.5 Defining a Data Type

We can do procedural abstraction of protocols along the same lines for actors. For example if we obtain the following actor protocol

```
[<when
        [[ ]]
        [<atomic>]
        [<when
                [[ ]]
                [<atomic>]
                [<when
                        [[ ]]
                        [<atomic>]>]>
        {when
                [[ ]]
                [<atomic>]
                [<when
                        [[ ]]
                        [<atomic>]
                        [<when
                                [[ ]]>
                         {when
                                [[ ]]
                                [<atomic>]} ]>]} ]>]
```

Then by identifying equivalent nodes we obtain the actor expr where

```
<define expr <actor [ ]
        <when
                [[ ]]
                [<atomic>]
                {[<expr> {expr} ]]>>>
```

Goodstein has many inductive proofs of the the properties of recursive programs. John McCarthy was one of the first to popularize the use of recursion induction for proving the properties of programs. The easiest way to do recursion induction is to provide at least one predicate for each recursive equation. Robert Floyd has proposed that predicates in the first order quantificational calculus be attached to the edges of flow charts in order to provide subgoals for proofs of properties of programs. In general we would prefer to proceed more constructively and to write intentions in PLANNER rather than in a form of the quantificational calculus. Finding an intuitionistic proof of a sentence in first order logic is the same problem as finding a recursive function that realize the the formula. Since the logistic system of PLANNER is very constructive, a proof of a PLANNER theorem entails being able to write the procedures which compute the values that identifiers in goals take on as a result of the goal being established. Intentions are a first step toward constructing models of the environment in which a process executes. We need to develop good ways to increase the expressive power of intentions. Currently the model of the computation must be expressed by intentions within the process being executed which makes it difficult to get a global view of the model of the execution of the process. The application of intentions in which we are most interested is their use to provide subgoals to enable us to deduce PLANNER theorems with loops in them. We shall say that an intention i characterizes a function f if whenever <f x> converges then <equal <f x> y> if and only if <i x y> is true. A long time ago John McCarthy and others proposed that the debugging problem be solved by proving that the procedure is correct once and for all. Using induction McCarthy and his students have proved that certain compilers are correct. The most important practical difficulty to the realization of the proposal is that for many functions f written in higher level languages it seems that all the intentions that characterize f are at least as long as f because the only way to tell whether the value of <f x> is correct or not is to do an equivalent computation all over again. A good example of such a function is eval in LISP. The function eval is an extreme example of a function that has no simple declarative input ouput characterization. A real challenge in automatic program writing is to develop a symbolic inetegration routine from the criteria that the derivative of the answer must be equivalent to the input. One approach toward constructing such a routine would be to make use of some results of Risch on what must be the form of the integrand as a function of the form of the integrand. In the case of the factorial function there are two obvious ways to compute the function: using recursion or using a loop. In other cases it is not so obvious how to find a sufficiently different equivalent program. We shall say that an intention i is implied by a function f if whenever <f x> converges then if <equal <f x> y>, then <i x y> is true. Implied intentions are useful when we are only interested in some property of the function and don't care to try to characterize it completely. For example we might not care whether a function that determines how to stack cubes always puts red cubes on the bottom of the tower that it is trying to build. Or we might be interested in proving that a scheduler for a time sharing system passes some test for fairness in its distribution of time to users. Another potential use for implied intentions is to provide subgoals to prove that a given function that uses lock and unlock and unlimited use of assignment in parallel computations is indeed determinate.

A more serious problem is that often we cannot develop reasonable implied overall intentions. Consider trying to write intentions for a chess program. We could require that the program play LEGAL chess but this is the least of considerations. How can we write intentions to the effect that the program should play GOOD chess? There is a completely trivial program which will play PERFECT chess given sufficient time and storage. However, the amount of time and storage required are wildly impractical. One might believe that the problem of writing overall intentions afflicts only game playing programs. However, the same problem arises in trying to write overall intentions for a robot. We can specify in detail a certain finite number of elementary procedures which the robot should be able to perform. In a given situation there may be some obscure way for the procedures to interract to provide a solution for a problem. However, it is not fair to blame the robot for not solving a very difficult problem. Thus we again have a problem writing realistic overall intentions.

7.2.3 Teaching Procedures by Deducing the Bodies of Canned Loops

If the type of control structure is known a priori, then the rest of the function can often be deduced. Often the control structure needed is a very commonly used loop such as the FOR loop in MATCHLESS, recursion on the tree structure of lists, or one of the loops in PLANNER such as TRY, FIND, or EXHAUST. We shall call loops such as the above "canned" loops since we will often pull them out and use them whole when we are in need of a control structure for a routine. The approach of using canned loops is the one used by Kleene for constructive realization functions for intuitionistic logic. Suppose that we know the following theorem about the predicate [REVERSE? x y] which means that y is the reverse of x. For example [reverse? aa aa] and [reverse? [1 2 [3 4]] [[3 4] 2 1]] are true. As before ' is used to suppress invocations, and a monad is defined to be an atom, a number, [ ], or [ ]. The function IDENTITY which is used below is the identity function.

```
<define th69 <consequent
        [a b c]
        [reverse? ?a ?b]
        <cond
           [<hasval? a>
                <cond
                    [&<goal !'<monad? .a>>
                        ;"if a is a monad then b should be equal to a"
                        <goal !'<is? .a ?b>>]
                    [¬"else"
                        <goal !'<not !'<monad? .a>>
                        <goal [reverse? !'<rest .a> _c]>
                        ;"otherwise let c be the reverse of the rest
of a"
                        <goal !'<is? ['(identity .c) !'<1 .a>] ?b>>]>]
            [¬"else" <fail>]>>>
```

We would like to find a function reverse such that [reverse? x [reverse x]] is always true. The theorem above suggests that we try to use linear induction on lists as the control structure. The schema for linear induction applied to the function reverse is:

```
<define reverse <function [x]
        !'<cond
           [!'<monad? .x>
                <tenprog [Y]
                        <assert !'<monad? .x>>
                        <goal [reverse? .x _Y]>
                       ;"find a Y which is the reverse of the monad
                              x and return it as value"
                        .Y>]
            [¬"else"
                <temprog [Y]
                        <assert !'<not !'<monad? .x>>>
                        <assert [reverse?
                                !'<rest .x>
                                !'<reverse !'<rest .x>>]>
                        <goal [reverse? .x _Y]>
                        .Y>]>>>
```

The above expression evaluates to the following definition:

```
<define reverse <function [x]
        <cond
            [<monad? .x>
                .x]
            [¬"else"
                [{identity <reverse <rest .x>>}
                <1 .x>]]>>>
```

7.2.4. Comparison of the Methods

Superficially considered, there is not much to be said about teaching procedures by telling. It is not always clear whether the procedure should be taught from the top down or the primitives should be taught first. However, the basics of the method are simple and direct. Unfortunately the teacher will not always know the code for the procedure which is to be taught. He might be engaged in wishful thinking hoping to find a procedure with certain properties. The method of canned loops is often applicable to such cases. Trying to use the method of canned loops has the problem that the control structure must be supposed. Often it is very difficult to guess the kind of control structure which will prove appropriate. Also the method of canned loops works on the problem in the abstract as opposed to specific examples where the identifiers are bound to actual values. The advantage of the abstract approach is that if it succeeds then the procedure will be known by its construction to have certain properties. On the other hand it is often easier to see what to do on concrete cases. Often it is easier to show someone how to do something than to tell him how to do it. Partly this is because the descriptive language necessary has not been adequately developed and so we use "body language". The approach of procedural abstraction is to combine together several concrete cases into one supposed general procedure. Properties of the general procedure must then be established by separate argument. If the protocols of the examples are produced by a goal-oriented language such as PLANNER, then there will be points along the protocols where certain predicates are known to be true. The predicates express the fact that some goal was established as true at that point. Often it is possible to show by mathematical induction that the corresponding properties in the abstracted procedure are always true when the procedure passes through the points. In this ay a problem solver can have a partial model of his problem solving procedures. The models can be expressed naturally in PLANNER. Also the method of procedural abstraction has the advantage that the control structure does not have to be supposed in advance. Often a problem solver will have the basic problem solving ability to solve any one of a certain class of problems. But he will not know that he has the capability. Writing a procedure which can be shown to solve the class enables the problem solver to bootstrap on his previous work. Procedural abstraction itself is further evidence for the Principle of Procedural Embedding. To implement the principle as a research program requires a high level goal-oriented formalism. PLANNER and some embellishments that w have made to the language are first steps toward realizing the Principle of Procedural Embedding.

7.3 Current Problems and Future Work

Currently we have mechanisms to handle the following kinds of "bugs" or "local changes" in programs:

MISIDENTIFICATION of NODES: If two nodes of a protocol have been mistakenly identified as being the same then the mistake can be corrected from new protocols which distinguish the nodes.

VARIABALIZATION: Procedures can be made more general by changing some of their constants into variables.

PATCHES: Existing routines can sometimes be converted into the desired procedure by introducing new intentions into them. The patch is produced by the code generated by the new intention as it is evaluated by INTENDER in the environment in which it was placed. Of course a bug is suspected at the point where an ordinary intention cannot be verified.

We need to find ways to improve the existing mechanisms and to find ways to handle other kinds of bugs and local changes. Also procedural abstraction must be generalized to accept higher level protocols and to make better use of existing procedural knowledge in doing the abstraction.

8. More Comparative Schematology

Abstract

Schemata are programs in which some of the function symbols are uninterpreted. In this chapter we compare classes of schemata in which various kinds of constraints are imposed on some of the function symbols. Among the classes of schemata compared are program, recursive, backtrack, and parallel.

8.1. Analytic Theory

8.1.1 Classes of Schemata

8.1.1.1 Recursive Schemata

The following is an informal progress report of some work that I have done with Mike Paterson. John L. White made important suggestions and corrections. The result that recursive schemata are more powerful than program schema was obtained as a term project in the spring of 1969. Rigorous proofs are not given here but just an indication of how a proof would go. Program schemata are nonrecursive procedures that have uninterpreted function symbols and predicate symbols. We shall use capital letters to denote uninterpreted symbols. We assume that within each computing domain that there is a distinguished element denoted by false and that all other elements of the computing domain are regarded as true in conditional expressions. Thus we do not need to distinguish between predicates and other functions. Iteration within program schemata is performed by REPEAT loops. Repeats are defined so that (repeat <body>) will repeatedly execute <body> until a (return <values>) statement is encountered at which point control is transfered out of the smallest enclosing block with the indicated values. Blocks can be given names and the function (exit <name> <values>) will cause control to leave the named block with the appropriate values. It is easy to see that any program schema in the sense of Patterson can be written using REPEAT and EXIT with out the use of GO. Writing iterative computations using REPEAT and EXIT has the advantage that all the loops are of necessity nested. We shall allow schemata to use a finite number of distinguished objects which can be tested by the binary predicate IS. For example (is x "hello") is true only if x is the distinguished constant "hello". Functions evaluate their arguments from left to right.

The following is an example of a program schema:

```
(g x) = (repeat ((y <- x))
        ;"y is a a register of the program schema g which is
initialized to the value of the argument x"
        (if (or (P x) (is x "dolly")) then (return y))
        (x <- (L y))
        (y <- (R (R y))))
```

The BNF syntax for program schemata is as follows:

```
        program ::= <term>
        term ::= <block> |
                <repeat> |
                <again> |
                <exit>
                (if <term> then <terms> else <terms>) |
                <assignment> |
                false |
                <literal-string> |
                <identifier> |
                <function-call>
        block ::= (block <body>)
        assignment ::= (<identifier> "<-" <term>)
        repeat ::= (repeat <body>)
        function-call ::= (<uninterpreted-function> <arguments>) |
                (is <term> <term>) |
                (call
                        (<uninterpreted-function> <arguments>)
                        <function>)
        again ::= (again) | (again <name>)
        exit ::= (exit <name> <terms>) | (return <terms>)
```

```
body ::= <name> <declaration> <terms> |
         <declaration> <terms>
terms ::= <term> | <term> <terms>
declaration ::= (<identifiers>)
arguments ::= | <terms>
identifiers ::= | <identifier> <identifiers>
```

A recursive schema is a program schema that is allowed to call itself or other recursive schemata recursively. The following is an example of a recursive schema k which is defined by a set of recursive equations:

```
(k x) = (if (P x) then x
                  else (C (k x) (m (R x))))

(m y) = (if (P (R y)) then (L y)
                  else (C (m (l y)) (k (k x))))
```

For any recursive schema defined by a set of recursive equations we can construct an equivalent recursive schema with only one equation and one additional argument to tell which equation is being simulated. This is possible because we allow recursive schemata to use a finite number of distinguished constants and predicates to test for these constants. The following is an example of a recursive schema that uses the interpreted constant symbols true and false.

```
(f x) = (if (P x)
           then
              (if (Q x)
                    then true
                    else false)
        elseif (f (L x))
                    then true
        else (f (R x)))
```

8.1.1.1.1 Comparison with Program Schemata

In fact the above recursive schema is not equivalent to any program schema. By equivalent we mean that the two schemata must both fail to terminate or both must return the same value for all interpretations of the functions P, Q, L, and R. Often we will take the set of uninterpreted terms as our domain of interpretation. In the above case the domain of interpretation is x (L x), (R x), (L (L x)), (L (R x)), (R (L x)), etc. The function letters L and R are interpreted as l and r where:

(l y) is defined to be the term (L y)

(r y) is the term (R y)

Thus (l (R (L x))) is the term (L (R (L x))). Two schemata are equivalent if and only if they define the same function on the domain of terms.

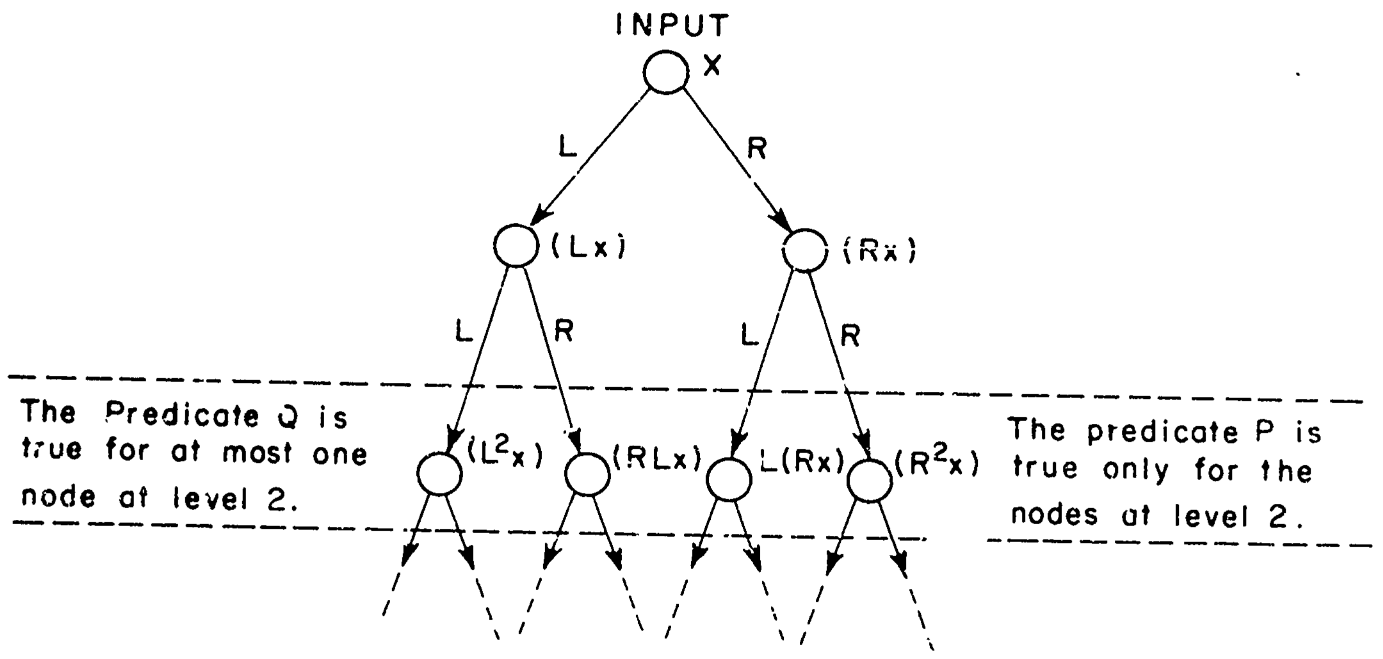
Theorem:

The function f defined above is not equivalent to any program schema.

Proof: Consider the following class of interpretations {I n} where n is a non-negative integer:

The domain of interpretation is the set of terms that can be constructed from the indeterminate x and the predicate letters L and R. The predicate Q is interpreted as a function q with range {true false}. The predicate P is interpreted as the function p:

(p (h//0 ...(h//m x)...)) = true for m = n


INPUT
X
L
R
(Lx)
(Rx)
L
R
L
R
(L²x)
(RLx)
L(Rx)
(R²x)
The Predicate Q is true for at most one node at level 2.
The predicate P is true only for the nodes at level 2.

COMPLETE L-R TREE FOR $\{I\ 2\}$

= false otherwise

where each h//i ("h subscripted by i") is the interpretation for R or the interpretation for L and there is at most one path such that

(q (h//0...(h//n x)...)) = true

The domain of {I n} is the set of all terms that can be constructed from the indeterminate x and the functions L and R. We are going to prove that for any program schema P we can find an integer t such that P does not define the same function as the recursive schema f on at least one member of the class {I n}. In the the interpretation {I 3}, we have the following L-R tree (where each node is a term in the domain of {I 3}):

```
{x
        {(L x)
                {(L (L x))
                        {(L (L (L x)))}
                        {(R (L (L x)))}}
                {(R (L x))
                        {(L (R (L x)))}
                        {(R (R (L x)))}}}
        {(R x)
                {(L (R x))
                        {(L (L (R x)))}
                        {(R (L (R x)))}}
                {(R (R x))
                        {(L (R (R x)))}
                        {(R (R (R x)))}}}}
```

The function p is true only on the right-most (i.e. bottom) nodes and q is true on at most one of the right-most (bottom) nodes. We shall define the state of a program schema P at a point in its computation to be the contents of the registers of P together with the statement of P that will be executed next. Two states S1 and S2 of P under the interpretion I will be said to be EQUIVALENT if p executes exactly the same sequence of instructions when started from S1 as when started from S2. We shall define the number of statements of a program schema to be the total number of left parentheses in the text of the program schema. Suppose we have a program schema P with s statements and k registers. In the interpretation (I n), the program schema P has at most s*((n+2)¬k) equivalence classes of states where ¬ is the exponential function. (Intuitively the only thing the schema can do is to count down each of its k registers to the bottom of the L-R tree and test each of them to see if it has reached the bottom.) However, a program schema needs at least 2¬n steps in order to check if q is true on each of the nodes at level n. But after 2¬n steps, P must be in an infinite loop since it will have arrived at two distinct nodes of the L R tree in the same equivalence class of states. To see the matter somewhat differently look at the sequence of equivalence classes of states. If the sequence repeats then the program schema is in an infinite loop. But the program schema must seek and test all 2¬n terminal nodes and then halt. Therefore the program schema needs at least 2¬n equivalence classes.

The Single Instance Theorem:

A single recursive schematic equation that defines a function form f can be transformed into an equivalent program schema if the form f appears only once in the definition of the function.

Proof:

Define (F¬n x) to be F applied n times to some argument x.

```
(F¬0 x) = x

(F¬(n+1) x) = (F (F¬n x))
```

For example (F¬1 x) is (F x) and (F¬2 x) is (F (F x)).

Suppose the definition of f is of the form

```
(f k) =  if (alpha k)
              then (beta k)
              else (gamma (f (delta  k)) k)
```

where (alpha k) is the expression that is evaluated before the recursive call to f, (beta k) is the expression that is evaluated if there is no recursive call to f, and (gamma (f (delta k)) k) is the value for a recursive call to f. The reader may or may not want to examin the following tree which shows f partially expanded:

```
(if (alpha (delta¬0 k))
     then
        (beta (delta¬0 k))
     else
        (gamma
                (if (alpha (delta¬1 k))
                     then
                        (beta (delta¬1 k))
                     else
                        (gamma
                                (if (alpha (delta¬2 k))
                                     then
                                        (beta (delta-2 k))
                                     else
                                        ...)
                                (delta¬1 k))
                (delta¬0 k)))
```

The function f can be re-written as follows:

```
(f k) = (block ((m <- k) n i j)
        ;"m, n, i, and j are registers of the program schema f; m is
initialized to the value of k"
        (repeat ()
             (if {alpha m} then (return))
                (m <- {delta m}))
        (i <- k)
        (n <- {beta m})
        (repeat ((i <- k) (n <- k))
                (if {alpha i}
                     then
                         (exit f n))
                (i <- {delta i})
                (repeat ((j <- i) (m <- k))
                         (if {alpha j} then (return))
                         (j <- {delta j})
                         (m <- {delta m}))
                (n <- {gamma n m})))
```

We would like to repeat the iterative definition of f giving comments. An expression that appears within [and] is an intention that is expected to be true whenever control passes through the expression. It is not necessary to understand the intentions in order to understand the schema f. In fact many readers might prefer not to read the intentions. The intention functions fa, fc, and fd are intended to express what goes on in loops a, c, and d respectively.

```
(f k) = (block ((m <- k) n i j)
        ;"m, n, i, and j are registers of the program schema f; m is
initialized to the value of k"
        (repeat a ()
             (if {alpha m} then (exit a))
                (m <- {delta m}))
        [define (fa m) = if {alpha m} then m else {delta m}]
        [(m = (fa k)) ;"It is our intent that m be equal to (fa k) at
this point.  It can be shown by induction that this intention is
always realized."]
        (i <- k)
        (n <- {beta m})
```

```
     (repeat c ((i <- k) (n <- k))
         (if {alpha i}
            then
              [(f k) = (fc {delta (fa k)} k) = n]
              (exit f n))
         [n = (f i)]
         [define (fd n m j) = if {alpha j} then {gamma n m}
else (fd n {delta m} {delta j}))]
         [define (fc n i) = if {alpha i} then n else (fc (fd n
k i) {delta i}))]
         [n = (fd {beta (fa k)} k i)]
         (i <- {delta i})
         (repeat d ((j <- i) (m <- k))
              (if {alpha j} then (exit d))
              (j <- {delta j})
              (m <- {delta m}))
         (n <- {gamma n m})
         [n = (f m)]))
```

8.1.1.1.2 Compilation

We can look at program schemata and recursive schemata as automata that operate on the universe of terms as a data space. A finite state schema automaton operates under a finite state control structure using a finite number of registers each of which can hold one term. As a primitive operation the automaton is allowed to create a term by applying a function to terms stored in its registers and then to store the result back in a register. In addition the automaton is allowed a finite number of primitive predicates to test the contents of its registers. The class of finite state schema automata is equivalent to the class of program schemata in the obvious way. Program schemata can be regarded as being executed by a finite state schema automaton after a suitable compilation. A pushdown schema automaton is defined to be a finite state schema automaton with a pushdown stack. In addition a pushdown schema automaton is allowed a finite number of distinguished constants as terms together with predicates that test for the distinguished constants. We will investigate the relationship between these machines and schemata. The appropriate kind of equivalence is one in which side effects are allowed. Two schemata will be said to be side-effect equivalent if they are the same function for all interpretations including those which involve side effects. An uninterpreted function may change the definition of any of the unintepreted functions as a side effect of being evaluated. For example the schemata j1 and j2 below are not side-effect equivalent.

```
(j1 x) = if (P x) then x
                  else (j1 (p1 (G x) (G x)))

(p1 x y) = x

(j2 x) = if (P x) then x
                  else (j2 (G x))
```

The free interpretations for side effect schemata are the ones in which each uninterpreted function symbol is interpreted as the function which evaluates to the list of all the primitive terms that have been previously evaluated in the computation. For example the side-effect protocol tree for j2 is

```
if (P x)
     then {x →(P x)}
     else
        if (P (G x))
             then {(G x) → [illegible] (G x) →(P x)}
             else
```

```
                    if (P (G¬2 x))
                         then {(G¬2 x) +(P (G¬2 x)) (G¬2 x) -(P (G x)) (G
x) -(P x)}
                         else...
```

On the other hand the side-effect protocol tree of j1 is:

```
if (P x)
     then {x +(P x)}
     else
        if (P (G x))
             then {(G x) +(P (G x)) (G x) (G x) -(P x)}
             else
                if (P (G¬2 x))
                     then {(G¬2 x) +(P (G¬2 x)) (G¬2 x) (G¬2 x) -(P (G
x)) (G x) (G x) -(P x)}
                     else...
```

Thus j1 and j2 are not side-effect equivalent.

Theorem: Side-effect equivalence is decidable for program schemata.

Proof: The proof is by tree expansion. Two program schemata are side effect equivalent if and only if for every execution path of one schema there is an execution path for the other with the uninterpreted functions called in exactly the same order. Given a cycle in one schema it is decidable whether the cycle can be embedded in the other.

Conjecture: Side-effect equivalence is decidable by tree expansion for recursive schemata.

If this conjecture is correct then we can attach a post processor to a compiler w[illegible] decides whether or not the compiled code is side-effect equivalent with the source code.

The BNF syntax for program schema automata is as follows:

```
program ::= <command>
command ::= <block> |
        <repeat> |
        <again> |
        <exit> |
        <push> |
        <pop> |
        <conditional> |
        <function-call>
value ::= false | identifier | <literal-string>
values ::= <value> | <value> <values>
pop ::= (pop) | (pop <identifier>)
exit ::= (exit <name> <values>) | (return <values>)
conditional ::= (iftrue <commands> else <commands>) |
        (ifempty <commands> else <commands>)
again ::= (again) | (again <name>)
push ::= (push) | (push <value>)
block ::= (block <body>)
repeat ::= (repeat <body>)
function-call ::=
                (call
                        <number-of-args>
                        <uninterpreted-function>) |
                (call
                        <number-of-args>
                        <uninterpreted-function>
                        (<identfiers>)
                        <commands>) |
                (call 2 is)
identifiers ::= | <identifier> <identifiers>
body ::= <name> <declaration> <commands> |
        <declaration> <commands>
declaration ::= (declarers)
declarers ::= | (<identifier> <value>) <declarers>
```

There are a few non-obvious constructs in the above syntax. The expression (pop <identifier>) removes the top element from the stack and makes it the new value of the identifier. Arguments to function are passed on the stack and the results are returned on the stack.

The Compilation Theorem:

For every recursive schema there is a side-effect equivalent pushdown schema automaton.

Proof:

We shall show how to compile the schema f defined below:

```
(f x) = (block ((y <- (H x)))
        ;"y is a new local which is initialized to (H x)"
        (if (P x)
             then (K x y)
        elseif (and y (P (f x)))
             then (K y x)
             else (G (K y x) y)))
```

The compiled form is

```
(f x) = (block ((y false))
        (push x)
        (call 1 H)
        (pop y)
        (push x)
        (call 1 P)
        (iftrue
                (pop)
                (push x)
                (push y)
                (call 2 K)
                (return 1)
            else
                (pop)
                (push y)
                (iftrue
                        (pop)
                        (push x)
                        (call 1 f)
                        (call 1 p)
                        (iftrue
                                (pop)
```

```
                    (push y)
                    (push x)
                    (call 2 k)
                    (return 1)
               else
                    (pop))
          else
               (push y)
               (push x)
               (call 2 K)
               (push)
               (push j)
               (call 2 G)
               (return 1))))
```

8.1.1.1.3 Schemata with Resets

Tags can be thought of as identifiers which are bound at each activation level. By passing the activation as a parameter the level of activation can be immediately reset by executing a transfer of control through the activation. In order to obtain an equivalent machine, we can extend the instructions of the push down schema automaton by allowing them to store a pointer to the top element of the stack into one of the registers. The resulting class of machines is called the reset push down schema automata. If the stack is ever popped back past a location that is pointed to by a register then the automaton halts with an error. We found discussions with Mike Fischer helpful in analyzing schemata with resets.

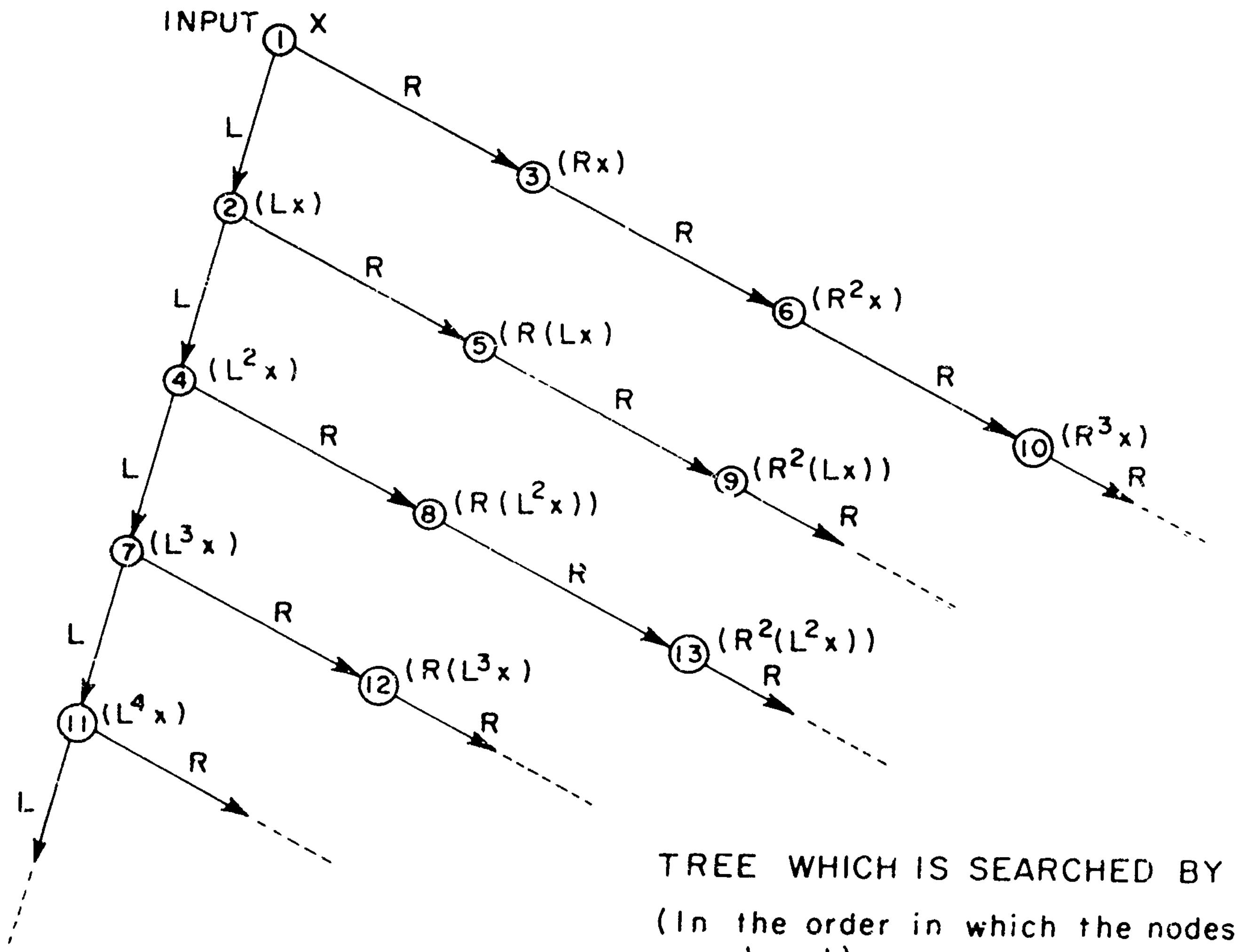
The Reset Theorem:

The class of reset push down schema automata is equivalent to the class of ordinary push down schema automata.

We shall show how we can translate a reset push down schema into an equivalent ordinary push down schema. An ordinary function call (f x//1 ... x//n) will be translated into (call (f x//1 ... x//n) (y y//1 ... y//1) body) where we will explain the body below. The idea is that if the function f wants to execute a non local transfer of control through argument x//i we can simulate this by returning the corresponding y//i as "exit" or "again" depending on whether the block x//i is to be exited or reiterated respectivey. Then the procedure which makes the call to f can test the values of the y//i and take the appropriate action depending how the name was generated. Consider the following example:

```
(try x)   = (repeat t1 ()
          (if (Q x)
               then
                   (x <- (F x))
          elseif (P x)
               then
                   (x <- (harder (F x) t1))
                   ;"the name t1 is an identifier"
                   (if (not x)
                         then (return false))
               else (return false)))

(harder x1 tag)  = (repeat ()
          (if (Q x1)
               then
                   (x <- (F x1)) ;"set the global x to (F x1)"
                   (again tag) ;"reiterate the repeat loop named tag"
          elseif (P x)
               then
                   (x1 <- (harder (F x1) tag))
                   (if (not x1)
                         then (return false))
               else (return false))
```

We can rewrite try and harder as try* and harder* respectively so that resets are eliminated.



TREE WHICH IS SEARCHED BY G

(In the order in which the nodes are numbered)

```
(try' x)   = (repeat t1 ()
        (if (Q x)
             then
                 (x <- (F x)))
        elseif (P x)
             then
                 (x <- (call (harder' (f x) t1) (y y1 y2)
                                (if
                                          (is y2 "again")
                                     then
                                          (again t1)
                                elseif
                                          (is y2 "exit")
                                     then
                                          (exit t1)
                                    else y)))
                 (if (not x)
                      then (return false))
             else (return false)))

(harder' x1 tag)   = (repeat ()
         (if (Q x1)
              then
                  (x <- (F x1))
                  (exit harder' false false "again")
                  ;"reiterate the loop named tag"
         elseif (P x1)
              then
                  (x1 <- (call
                                  (harder (F x1) tag)
                                  (y y1 y2)
                                  (if
                                            (is y2 "exit")
                                       then
                                            (exit harder' false false
"exit")
                                  elseif
                                            (is y2 "again")
                                       then
                                            (exit harder' false false
"again")
                                       else y)))
                  (if  (not x1)
                       then (return false))
              else (return false)))
```

8.1.1.1.4 Decompilation

The Decompilation Theorem:

For every push down schema automaton we can effectively construct a side-effect equivalent recursive schema.

Proof:

The only difficult constructs to translate are the push and pop commands. We shall translate (push <value>) as (<function> <value> tags false) where <function> is a unique function name distinct from all others. The function is defined to be have two arguments x and y and have a body which is the code that follows the push command which is being translated. The command (pop) is translated as (GO <tag>) <tag>: where <tag> is a unique tag distinct from all others. whereas (pop <identifier>) is translated as (<identifier> "<-" x) (GO <tag>) <tag>:. The idea is that there must be a tag for every instance of a call to pop so that control can get back to the proper place.

Consider the following push down schema automaton:

```
(f y) = (block ()
        (push)
        (call 1 g)
        (return 1))

(g y) = (repeat ()
        (push)
        (call 1 P)
        (iftrue
                (pop)
                (push)
                (call 1 Q)
```

```
                    (iftrue
                            (pop)
                            (repeat ()
                                    (ifempty
                                            (terminate "t"))
                                    (pop)))
                        else
                            (pop)
                            (pop)
                            (ifempty
                                    (terminate false))
                                else
                                    (call 1 R)
                else
                    (pop)
                    (push)
                    (call 1 L)))
```

The schema f can be decompiled as follows:

```
(f x) = (block outer ()
                (f0 x false false false false "t"))

(f0 x n1 n2 n3 n4 n5 n6 empty) = (repeat ()
        (f1 x t1 t2 t3 t4 t5 t6 false)
        (x <- (P x))
        (if
                x
            then
                (go n1)
            t1:
                (f2 x t1 t2 t3 t4 t5 t6 false)
                (x <- (Q x))
                (if
                        x
                    then
                        (go n2)
                     t2:
                        (repeat ()
                                (if
                                        empty
                                    then
                                        (exit outer "t"))
                                (go n3)
                            t3:
                                )
                    else
                        (go n4)
                    t4:
                        (go n5)
                    t5:
                        (if
                                empty
                            then
                                (exit outer false)
                            else
                                (x <- (R x))))
            else
                (go n6))
            t6:
                (f3 x t1 t2 t3 t4 t5 t6 false)
                (x <- (L x)))

(f1 x n1 n2 n3 n4 n5 n6 empty) = (block ()
        (if
                x
            then
                (go n1)
            else
                (go n2)))

(f2 x n1 n2 n3 n4 n5 n6 empty) = (block ()
        (if
                x
            then
                (go n3)
            else
                (go n4)))

(f3 x n1 n2 n3 n4 n5 n6 empty) = (block ()
        (x <- (L x))
        (f0 x n1 n2 n3 n4 n5 n6 empty))
```

8.1.1.1.5 Primitive Recursive Schemata

Definition a recursive schema f will be said to be PRIMITIVE RECURSIVE in the the uninterpreted function symbols U if f can be defined recursively as (f x//1 ... x//n) = phi[x//1 ... x//n] where each instance (f t//1 ... t//n) of a call to f within phi[x//1 ... x//n] has t//1 of the form (h x//1) where h is in the set U and the only other functions in the definition are either uninterpreted or are themselves primitive recursive in U.

For example the following schema is primitive recursive in {L R}.

```
(f x) = if (P x)
              then (Q x)
              else (C (S (L x)) (f (R x)))
```

The following schema is not primitve recursive in {S}:

```
(ackerman w x y) =
        (if (Z x)
            then
                if (Z w)
                    then y
                elseif (O w)
                    then (ZERO)
                else (ONE)
        elseif (Z w)
            then
                (P
                        (ackerman (ZERO) (S x 1) y)
                        (ONE))
        else
                (ackerman
                        (S w 1)
                        (ackerman w (S x 1) y)
                        y))
```

8.1.1.2 Schemata with Counters

We would like to present another example of a function that can be computed by a recursive schema but not by any program schema. Define (F¬n x) as in the proof of the Single Instance Theorem. Thus ((F¬n+1) x) = (F (F¬n x)). Suppose that we successively compute (F x), (F (F x)), etc. As we successively compute the quantity (F¬i x) for some integer i we shall keep a running count of the number of times that (P (F¬j x)) has been true for j less than i, minus the number of times that (P (F¬j x)) has been false for j less than i. If this count ever goes negative then we shall return false as the value of the function (zero x), otherwise the function (zero x) will run forever.

The Counting Conjecture for Program Schemata

The recursive schema 'zero' defined below is not schematically equivalent to any program schema.

```
(zero x) = (repeat a ()
             (if (P x)
                 then
                    (x <- (positive (F x)))
                    (if x
                         then
                            (again a))
                         else
                            (return false)
                 else
                    (return  false))
        end
```

The schema 'zero' uses the schema 'positive' to keep track of the count by the depth of recursion of the schema 'positive'.

```
(positive x) = (repeat a ()
                 (if (P x)
                      then
                         (x <-- (positive (F x)))
                         (if x
                              then
                                 (again a))
                              else
                                 (return false)
                      else
                         (return (F x)))
        end
```

Using the technique of loop elimination we can convert the above functions into purely recursive schemata. We shall define a schema zero1 which is equivalent to zero and a schema positive1 which is equivalent to positive.

```
(zero1 x)= (if (P x)
             then
                (if (positive1 (F x)))
                     then
                        (zero1 (positive1 (F x)))
                     else
                        false)
             else
                false)

(positive1 x)=
        (if (P x)
             then
                (if (positive1 (F x))
                     then
                        (positive1 (positive1 (F x)))
                     else
                        false)
             else
                (F x))
```

The protocol tree for the schema zero is

```
(if (P (F¬0 x))
     then
        (if (P (F¬1 x))
              then
                 (if (P (F¬2 x))
                       then
                          ...
                       else
                          (if (P (F¬3 x))
                                then
                                   ...
                                else
                                   (if (P (f¬4 x))
                                         then
                                            ...
                                         else
                                            false)))
              else
                 (if (P (F¬2 x))
                       then
                          (if (P (F¬3 x))
                                 then
                                 else
                                    (if (P (F¬4 x))
                                          then
                                          else
                       else
                          false))
     else
        false)
```

However a program schema can solve the problem if we give it a counter. We postulate the functions "+", "-", and zero? which respectively add, subtract, and test for zero. The following program schema is schematically equivalent the the function zero:

```
(zero1 x) = (block (n) (return (zero2 x)))
(zero2 x) = (repeat ()
                (if (P x)
                      then
                         (x <- (P x))
                         (n <- n + 1)
                    else
                         (if (zero? n) then (return false))
                         (n <- n-1)))
```

By allowing recursive schemata to use a counter, we can construct a function 'reczero' that is not equivalent to any ordinary recursive schema. the function reczero counts the number of nodes along the bottom of the L-R tree that have the property P minus the ones that do not have the property P. The function returns the value false if the count ever goes negative. We assume that arguments are evaluated from left to right.

The Counting Conjecture for Recursive Schemata:

The schema (with counters) reczero defined below is not equivalent to any ordinary recursive schema.

```
(reczero x) = (block (n) (return (reczero1 x)))

(reczero1 x) =
        (if (BOTTOM? x)
             then
                (if (P x)
                     then
                        (n <- n+1)
                        true
                     else
                        (if (zero? n) then (return false))
                        (n <- n-1)
                        true)
             else
                (if (not (reczero1 (L x))) then (return false))
                (if (not (reczero1 (R x))) then (return false))
                (return true))
```

The reason that reczero is not equivalent to any recursive schema is very similar to the reason that no recursive schema can search the branches of the L-R tree in parallel. If a recursive schema is equivalent to reczero then it is constrained to search the tree in essentially the same order that reczero searches the tree. Otherwise it could be made to fall into an infinite loop on an interpretation where reczero converges. We conjecture that constrained in this fashion a recursive schema has only a finite number of states in which to try to keep the count. The recursive schema cannot succeed for the same reason that we conjecture that no program schema is equivalent to the function zero defined above.

Conjecture: the following function is not schematically equivalent to any purely schematic recursive system of equations. The function even is supposed to test whether the number of bottom nodes of a L-R tree that are true for the predicate P is the same as the number that are false for the predicate P. The schema 'even' differs from the schema 'reczero' in the crucial respect that 'even' always looks at all the bottom nodes before it comes to any conclusions. Thus a recursive schema that tries to imitate the schema even has a lot more room in which to maneuver. We conjecture that no recursive schema can have enough internal states to be equivalent to the function even defined below.

```
(even x) = (block (n)
        (even1 x)
        (return (zero? n )))

(even1 x) =
        (if (BOTTOM? x)
             then
                (if (P x)
                     then
                        (n <- n+1)
                x)
            else
                (n <- n-1)
                x)
else
     (even (L x))
     (even (R x)))
```

8.1.1.3 Parallel Schemata

We introduce the delimiters "|(" and ")" to delimit quantities that are to be computed in parallel. Whenever a process executes an expression like |(x) it divides into two processes. One process executes x and the other attempts to continue normal execution. For example in the expression |(2+3)*(4*5), the product 4*5 is computed in parallel with the sum 2+3. Thus the expression "(block |(return x) (return y))" is defined to be the value of x or y depending on which evaluates first in some particular but unspecified parallel computation. Processes can coordinate their actions through locks. Any expression x can be locked by (lock x) provided that the expression is not already locked. If x is already locked then any process which executes (lock x) will be blocked until x is unlocked by the primitive (unlock x). However a process can execute (locked? x) which will return true is x is locked but will lock x if it is unlocked. The kind of call delimited by "|(" and ")" can be implemented using the following primitives:

(create f) will create a new processwhich will begin execution with a call to f and will return the name of the created process as the value of the function create.

(resume (p send-args) f) will suspend execution of the process that calls resume and will resume execution of the process named p with arguments send-args. If the process p is already running then the process which called resuume will be blocked until p becomes suspended. If the process which called resume is itself ever resumed then it will invoke f with the arguments received.

(fork (p send-args)) will resume exeuction of the process p with arguments send-args and in parallel return the name of the process forked as the value of the function fork.

(interrupt p x) will interrupt the execution of the process p and then begin execution of x IN THE PROCESS p.

(step p) will step the process p through one step.

By adding the above primitives we obtain the class of Parallel Schemata. It is our thesis that the class of Parallel Schemata is in fact UNIVERsAL for the class of all effective schemata. By this we mean that for any effective schema there is a timing side-effect equivalent parallel schema. Two automat a and b will be said to be timing side-effect equivalent if for every computation of a there is a side-effect equivalent computation of b where the timings of the control primitives of b are allowed to be arbitrarily adjusted and vice-versa.

We define the following function using parallel processing:

```
(f x)=(if (P x)
           then x
           else
              begin
                    |(return (f (L x)))
                    (return (f (R x)))
              end)
```

The above function is determinate (i.e. halts and has the same value independent of the relative speed at which the sub-processes run) on infinite binary trees in which the predicate P is true on only one node.

The Parallel Evaluation Theorem:

The function f defined is not equivalent to any recursive schema.

Proof: Suppose a set of recursive equations {f//0, f//1, ..., f//n} is schematically equivalent to f with f//0 equivalent to f. That is for all interpretations of the uninterpreted function symbols, the schemata f and f//0 are the same function. Suppose that we start up f//0 on input x and make the predicate P false for every node to which it is applied as f//0 computes along. If the computation converges then f//0 does not look at some node which is a contradiction of the supposition that f//0 is equivalant to f. Therefore the computation runs forever and the sequence of statements through which the control passes is ultimately periodic. Consider the sequence of arguments to one of the functions (call it f//i for "f subscripted by i") as the control passes through one cycle. Suppose that f//i is a function of j arguments: a//1,...,a//j. The arguments with which f//i will be called after the control has passed through one cycle are terms definable from a//1,...,a//j. Let us call them a//1¬1,...,a//j¬1. The situation can be diagrammed as follows:

.
.
.

. . . . (f//i a//1,...,a//j); the beginning of the cycle in the

```
control structure
                 .
                 .
                 .
        (f//i a//1¬1, ...,a//j¬1) ; We pass through the same point of
the cycle in the control structure
```

If none of a//1¬1, ...,a//j¬1) is the same as one of a//1,...,a//j then we are done since the arguments of the recursive equations are tracing j paths down an expontentially growing tree which means that some node is not looked at. If we set the interpretation so that P is true for the node then we have a contradiction. We conclude that the fact that one of a//1¬1, ...,a//j¬1 might be same as one of a//1,...,a//j is a nuisance. Let us call the arguments to f//i after we have gone through the cycle k times a//1¬k,...a//j¬k. Observe that if we go through the cycle j! times then there will be some i such that i is less than j! and a//1¬i,...,a//j¬i has the property that it is an epicycle. By this we mean that some a//q¬i is the same as one of a//1,...,a//j if and only if it is the same as a//q. All such a//q do not contribute to the number of nodes examined since they are repeats of nodes previously examined in exactly the same way. The situation can be diagrammed as follows:

```
                 .
                 .
                 .
        (f//i a//1,...,a//j);
                 .
                 .
                 .
        (f//i a//1¬1, ...,a//j¬1)
                 .
                 .
                 .
```

(f//i a//1¬k, ...,a//j¬k); the beginning of the epicycle in the control structure

.
.
.

(f//i a//1¬(2*k), ...,a//j¬(2*k)); we pass through the same point in the epicycle

Threrefore we can complete our proof by applying to epicycles the above argument that we used for cycles.

8.1.1.4 Locative Schemata

The Locative Theorem:

If locations of identifiers are an allowed data type, then the control structure of recursive schemata can compute any partial recursive function.

Proof:

Let (at x) denote the location of the identifier x. Furthermore suppose that we have a function in of one argument which will return the contents of its argument. The proof will be phrased in terms of pushdown schema machines. We can define a counter using a register as follows:

```
(block ((c1 false))
        (initialize-counter1) =(block ((w false))
                (push (at w))
                (pop c1))
        (count-up1) = (block ((y false))
                (push c1)
                (pop y)
                (push (at y))
                (pop c1))
        (count-down1) = (block ()
                (push c1)
                (call 1 in)
                (pop c1))
        (zero-test1) = (block ()
                (push c1)
                (call 1 in)
                (iftrue (push "t")) else (push false))))
```
Marvin Minsky proved that two counters are universal. Q.E.D.

8.1.1.5 Schemata with Selectors and Replacement

Another way in which we can proceed is to impose data types on the computing domain. Storage off the stack can be established by postulating a constructor c and selectors s1 and s2 such that for all x and y in the computing domain we have:

```
(s1 (c x y)) = x
(s2 (c x y)) = y
```

in the domain of interpretation. Classically we would postulate that every call to the constructor must return a new element of the computing domain.

8.1.1.6 Schemata with Free Variables

```
(c x y) = (block (z)
                 (z <- (s1 free-storage-list))
                 (free-storage-list <- (s2 free-storage-list))
                 ;"free-storage-list is free in c"
                 (return (CONSTRUCTOR x y z)))
```

The point is that in general (c x y) will not be the same as (c x y) because of use of assignment on the free varialbe free-storage-list. Other than in this fairly trivial way, schemata do not add any power to recursive schemata.

8.1.1.7 Schemata with Equality

Schemata with equality are allowed to make use of a special predicate (= x y) whose interpretation is that x and y are the same element of the domain of interpretation. Universal domains of interpretation for schemata with equality are the Herbrand universe with a congruence relation theta such that:

```
1: theta is an equivalence relation

2: if x/1 theta y/1, ..., and x/n theta y/n then for each
uninterpreted function f and predicate p:
        (f x/1... x/n) theta (f y/1 ... y/n) and
        (p x/1 ... x/n) if an only if (p y/1 ... y/n)
```

In other words the elements of the domain of interpretation are the equivalence classes of theta.

8.1.1.8 Hierarchical Backtrack Schemata

PLANNER uses a more powerful control structure than that of the recursive function call. A BACKTRACK CONTROL STRUCTURE is used which means that at any point a process can fail which will cause it to back up to some previous state and then continue. The primitive function (FAIL) will generate a simple failure. The primitive function (FAILPOINT try lose) will evaluate the expression try. If the evaluation succeeds then the value of the function FAILPOINT is the value of try. Otherwise the value of the function FAILPOINT is the value of lose. For example the value of

```
        (+
                (failpoint (x <- 2) (x <- 3))
                (if x=2 then (fail) else 4))
```

is 7 since x first gets the value 2 but then is given the value 3 when a failure backs up to the function FAILPOINT.

8.1.1.8.1 Comparison with Recursive Schemata

We shall give an example to show that backtrack control structure is more powerful than recursive control structure.

Backtrack Schemata Are More Powerful than Recursive Schemata

The backtrack schema g defined below is not equivalent to any recursive schema. What the schema g does is to search the following tree for x looking for a node on which P is true:

```
x
        (L¬1 x)
                (L¬2 x)
                        (L¬3 x)
                        (R¬1 (L¬2 x))
                                (L¬4 x)
                                (R¬1 (L¬4 x))
                (R¬1 (L¬1 x))
                        (R¬2 (L¬1 x)))
        (R¬1 x)
                (R¬2 x)
                        (R¬3 x)
```

We have shown in the section on parallel schemata that no recursive schema can do the search.

```
(g x) = (h (f x))

(h z) = (if z
            then
               "true"
            else
               (fail))

(f x)=
    (fail?
        (P x)
        (block (y)
                ;"y is a new local"
                (y <- x)
                (k
                        (f (L x))
                        (if (P y)
                             then true
                             else (y <- (R y) false)))))
```

The reason that we make the function k defined below into a separate function is so that BOTH arguments will be evaluated.

```
(k s t) = if s
            then
               "true"
            else t
```

Proof: The procf is similar to the proof of the parallel evaluation theorem. Suppose a set of recursive equations {f//0, f//1, ..., f//n} is schematicaly equivalent to f with f//0 equivalent to f. Suppose that we start up f//0 on input x and make the prdicate p true for every node to which it is applied as f//0 computes along. If the computation convereges then f//0 does not lock at some node which is a contradiction of the supposition that f//0 is equivalent to f. Therefore the computation runs forever and the sequence of statements therough which the control passes is ultimately periodic.

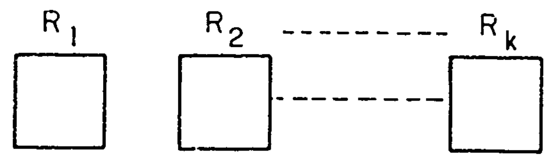
8.1.1.8.2 Comparison with Multiprocess Schemata

The method by which multiprocess schemata can simulate hierachical backtrack schemata is messy but straight forward.

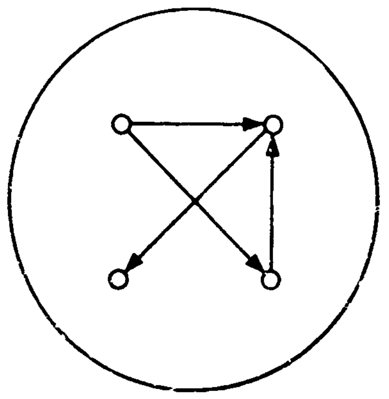
Multiprocess schemata are more powerful than backtrack schemata. One example which may show this is the one used to show that parallel schemata are more powerful than recursive schemata. Unfortunately we have not yet been able to prove that backtrack schemata cannot search the full L-R tree. So we shall resort to brute force techniques.

We would like to define the P-length of an expression x as the number of times which D can be applied to x before (P x) is true.

PROGRAM SCHEMATA



k REGISTERS EACH OF WHICH CAN HOLD AN INTEGER UP TO n



s STATEMENTS

HAS AT MOST sn^k STATES

376-a

Thus (P-length x) = (if (P x) then 0 else 1+(P-length (D x))) Now we would like to define a schema expt such that

(expt x y) = (I¬(2¬(P-length x)) y)

Suppose that (P-length x) = 2. Then (expt x y) = (I¬(2¬2) y) = (I¬4 y) = (I (I (I (I y)))).

```
(expt x y) = (if (P x)
               then
                   (I y)
               else
                   (expt (D x) y (expt (D x) y)))
```

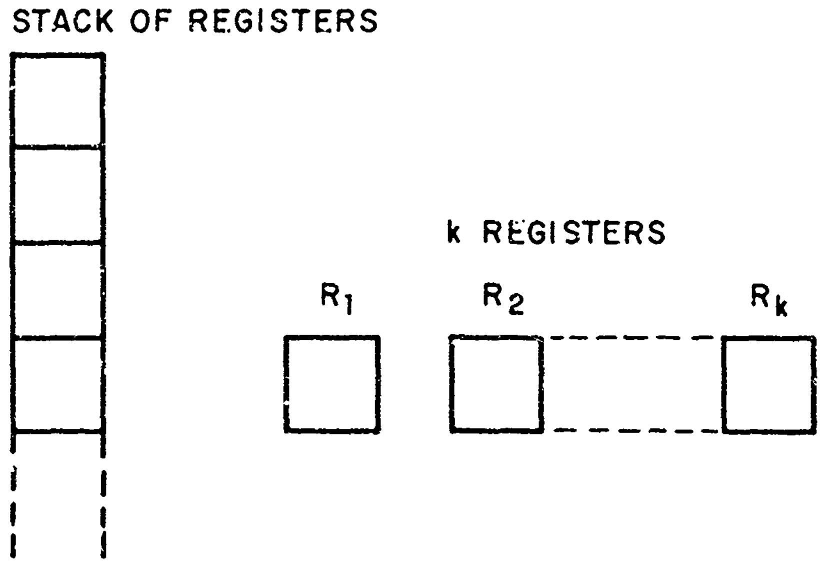
Now we claim that there is no program schema which is equivalent to expt. Suppose to the contrary that there is a program schema with k registers and s statements which is equivalent to expt. Such a program schema has at most only s * k¬(P-length x) equivalence classes of states. Thus if it runs for more than s * k¬(P-length x) steps it must be in a loop. Therefore it cannot possibly produce the output (I¬(2¬(P-length x)) y) since s * k¬(P-length x) is less than 2¬(P-length x) for large values of (P-length x). This is a contradiction.

In an exactly analogous fashion we can prove that there is no recursive schema expt2 such that

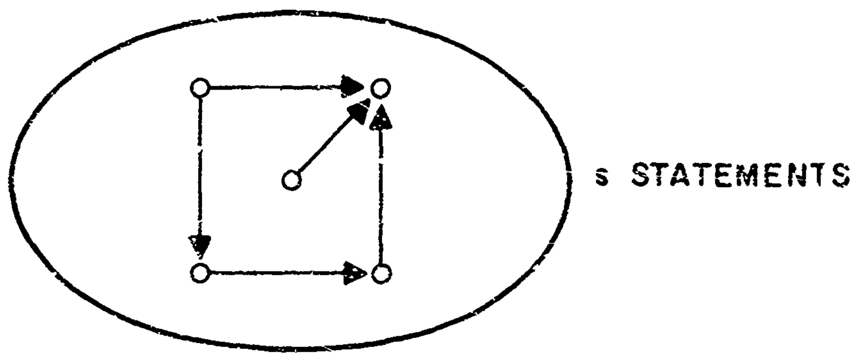
(expt2 x y) = (I¬(2¬(2¬(P-length x))) y)

Suppose that there is a recursive schema with k registers and s statements which is equivalent to expt2. Such a recursive schema has at most only

RECURSIVE SCHEMATA



EACH REGISTER CAN HOLD AN INTEGER UP TO n.



HAS AT MOST $sn^k\ n^{sn^k}$ STATES

J = s * (P-length x)¬k * (P-length x)¬(s * (P-length x)¬k) equivalence classes of states. The same state counting argument shows the contradiction. The above argument has been independently discovered by Robin Milner.

Theorem: Multiprocess schemata are more powerful than backtrack schemata

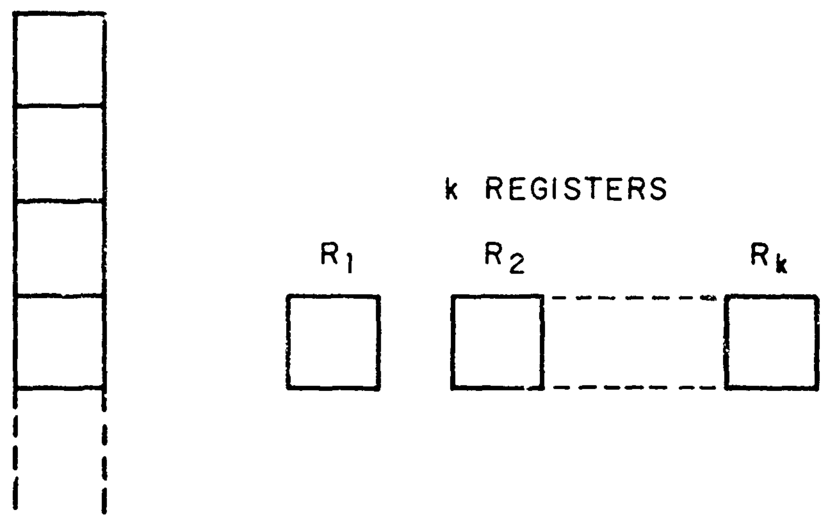
Proof: We will apply our brute force technique. There is no backtrack schema expt3 such that

(expt3 x y) = (I¬(2¬(2¬(2¬(P-length x)))) y)

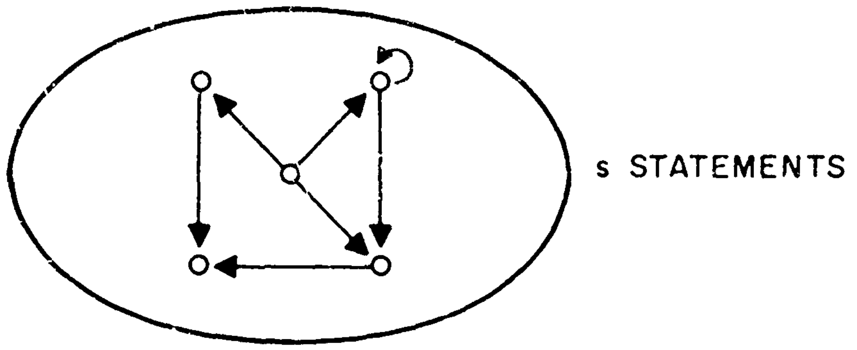
Suppose that there is a backtrack schema with k registers and s statements which is equivalent to expt3. Let J be as defined above. The recursive schema has at most J¬J equivalence classes of states. Thus if it runs for more than J¬J steps it must be in a loop. Therefore it cannot possibly produce the output (I¬(2¬2¬2¬(P-length x)) y) since J¬J is less than 2¬2¬2¬(P-length x) for large values of (P-length x). This is a contradiction.

BACKTRACK SCHEMATA

STACK OF REGISTERS



EACH REGISTER CAN HOLD AN INTEGER UP TO n



HAS AT MOST J^J STATES WHERE

$$J = sn^k \; n^{sn^k}$$

8.2. Synthetic Theory

8.2.1 Realizations

8.2.1.1 Realizations for the Quantificational Calculus

We would like to show how we can use schemata to express procedurally the meaning of certain constructive logically valid sentences in the predicate calculus. Classically, intuitionistic logic has been used to prove constructive sentences. However, the connection between this language and push down schema automata is somewhat indirect. We need to define the notion of a schema g realizing a formula phi. Roughly speaking g realizes phi if it tells how to compute the value of phi from the subformulas of phi depending on the logical connectives of phi. Kleene's notion of "g realizes phi" is defined by induction on the structure of phi:

For {terms}. g realizes phi where phi is a term if g is true if and only if phi is true. For example (P (F w) z) realizes (P (F w) z).

For {and...}. g realizes phi = (and theta psi) if (g 0) realizes theta and (g 1) realizes psi. Note that g really is two functions in disguise.

For {or...}. g realizes phi = (or theta psi) if whenever (g 0) is false then (g 1) realizes psi and whenever (g 0) is not false then (g 1) realizes theta.

For (implies...). g realizes phi = (implies theta psi) if whenever h realizes theta then (g h) realizes psi.

For (not...). g realizes phi = (not theta) if for no h is it the case that (g h) realizes theta.

For (all...). g realizes phi = (all x [theta x]) if for all x it is the case that (g x) realizes [theta x].

For (some...). g realizes phi = (some x [theta x]) if (g 1) realizes [theta (g 0)].

Consider the following formula which we shall call phi:

```
(implies
        (some x
                (implies (A x) (B x)))
        (implies (all x (A x)) (some x (B x))))
```

We claim the function g defined below realizes phi.

```
g = (lambda h (lambda k (lambda s
        (if s = 0
             then (h 0)
             else ((h 1) (k (h 0)))))))

        Suppose that h realizes (some x (implies (A x) (B x)))
        (h 1) realizes (implies (A (h 0)) (B (h 0)))
                suppose that k realizes (all x (A x))
                (k (h 0)) realizes (A (h 0))
                ((h 1) (k (h 0))) realizes (B (h 0))
                (((g h) k) 1) realizes (B (((g h) k) 0))
                ((g h) k) realizes (some x (B x))
        (g h) realizes (implies (all x (A x)) (some x (B x)))
g realizes phi
```

We are interested in knowing when a formula can be realized constructively.

Realization Theorem for Recursive Schemata with Functional Arguments.

If phi is proveable in intuitionistic logic, then phi is realizable by a recursive schema with functional arguments. The Realization Theorem represents one approach toward a constructive theory of computation. From a description of the kind of object that we would like to have given the description of certain other objects as input, we derive a program for computing our goal. Actually we shall prove that for intuitionistic logic the realization function can be made primitive recursive. The proof is a slight modification of the standard proof for the integers. It is a warm up for the analogous proof for the deductive system of PLANNER. However, in PLANNER we require the full power of the recursive functions for our constructive realizations.

Proof: The following proof is by induction on the structure of intuitionistic proofs using natural deduction. It goes by straightforwardly winding and unwinding of definitions. With a little work we could get PLANNER to create the proof.

```
{and introduction}
        theta realized by say g
        psi realized by say h
        ----------
        (and theta psi) realized by (lambda s (if (s = 0) then g else
h))
```

```
{and elimination}
        (and theta psi) realized by say g
        ----------
        theta realized by (g 0)
        psi realized by (g 1)

{or intro}
        psi realized by say g
        ----------
        (or theta psi) realized by (lambda t (if t=0 then false else
g))
        (or psi theta) realized by (lambda t (if t=0 then true else
g))

{or elim}
        (or theta psi) realized by say g
                theta hypothesis; suppose that theta is realized by h
                                .
                                .
                                .
                eventually deduce say omega which is realized by (m h)
for some recursive m using the inductive hypothesis
                psi hypothesis; suppose the psi is realized by k
                                .
                                .
                                .
                eventually deduce omega which is realized by (l k) for
some recursive l using the inductive hypothesis
        ----------
        omega which is realized by (if (g 0) then (m (g 1)) else (l (g
1)))

{implies intro}
                omega hypothesis; suppose omega is realized by h
                                .
                                .
                                .
                eventually deduce say psi which is realized by (g h)
for some recursive g using the inductive hypothesis.
        ----------
        (implies omega psi) realized by (lambda h (g h))

{implies elim}
        (implies omega psi) realized by say g
        omega realized by say h
        ----------
        psi realised by (g h)
```

```
{neg intro}
                omega hypothesis; suppose that omega is realized by h
                                 .
                                 .
                                 .
                eventually deduce say (not psi) which is realized by
(g h) for some recursive g using the inductive hypothesis
                eventually deduce psi which is realized by (k h) for
some recursive k using the inductive hypothesis.
        ----------
        (not omega) which is realized by any function since it is
impossible for both (not psi) to be realized by (g h) and for psi to
be realized by (k h).

{all intro}
                x|
                 |
                 |
                                 .
                                 .
                                 .
                 |
                 |eventually deduce say [omega x] which is realized by
(g x) for some recursive g using the inductive hypothesis
        ----------
        (all x [omega x]) realized by (lambda x (g x))

{all elim}
        (all x [omega x]) realized by say g
        ----------
        [omega t] for some term t; realized by (g t)

{exist intro}
        [omega t] is realized by say g where t is a term
        ----------
        (exist x [omega x]) is realized by (lambda s (if (s = 0) then
t else g))

{exist elim}
        (some x [omega x]) realized by say g
                x|[omega x] realized by (g 1)
                 |
                 |
                                 .
                                 .
                                 .
                 |
                 |eventually deduce say psy which does not contain x
free;  psy is realized by (m (g 0) (g 1)) for some recursive m using
the inductive hypothesis.
```

----
psy

Thus we have completed the inductive proof.

Intuitionistic Implementation Theorem

For every recursive schema P, we can effectively find a first order formula [theta x y] such that P is total if and only if (all x (some y [theta x y])) is proveable in intuitionistic logic. Furthermore, the program P on input x converges to the value y if and only if [theta x y] is proveable in intuitionistic logic. We assume that all uninterpreted function symbols in schemata are total.

We shall give an example of how to construct the formula theta for the following program which is due to Paterson:

```
(g x) = (if (T (F x))
            then (h x (F x))
            else x)

(h x y)=
        (if (T (F (F y)))
            then x
        elseif (T (F x))
            then (h (F x) (F (F y)))
        else (g (F x)))
```

We can obtain the formula that we require by doing a straight forward translation of the recursive equations into the quantificational calculus. These formulas are similar in intent to those of Manna, however we need use only intuitionistic logic to obtain the result we require. The formula [theta x y] to be constructed is the conjunction of the following three formulas where "iff" is an abbreviation for "if and only if":

```
(iff
        (PG x y)
        (or
                (and (T (F x)) (PH x (F x) y))
                (and (not (T (F x)) (y = x)))))

(all x1 x2 y (iff
        (PH x1 x2 y)
        (or
                (and (T (F (F x2))) (y = x1))
                (and
                        (not (T (F (F x2))))
                        (T (F x1))
                        (PH (F x1) (F (F x2)) y))
                (and
                        (not (T (F (F x2))))
                        (not (T (F x1)))
                        (PG (F x1) y)))))

(all x (or (T x) (not (T x))))
```

The last statement comes from the fact that we are assuming that all uninterpreted functions are total. The schema g is indeed total.

Even after adding selectors and constructors the realization theorem can still be proved in the standard way. We introduce the predicate atom which tests to see if its argument is atomic and thus cannot be broken down using the selectors. The following rule is added to intuitionistic logic:

```
        (all x (implies (atom x) [theta x])) realized by say g
                x,y|[theta x] hypothesis; suppose [theta x] is
realized by (m x)
                    |[theta y] hypothesis; [theta y] is realized by (m
y)
                    |
                        .
                        .
                        .
```

```
                                   |
                                   |eventually deduce [theta (c x y)] realized by say
(h m x y) using the inductive hypothesis
          ----------
         (all x [theta x]) realized by
(k x) = (if (atom x)

              then (g x)

              else (h k (s1 z) (s2 z)))
```

Sometimes an increase in efficiency can be obtained from replacement operators r1 and r2 such that

if x = (s1 z) and y = (s2 z) then after (r1 z w) we have (s1 z) = w, and (s2 z) = y
if x = (s1 z) and y = (s2 z) then after (r2 z w) we have (s1 z) = x, and (s2 z) = w.

We shall call schemata that allow the use of selectors and replacement operators list structure schemata. Two schemata will said to be equivalent as list structure schemata if for all interpretations of the uninterpreted function symbols they are the same function. For schemata that do not explicitly contain s1, s2, r1, or r2 list structure equivalence is the same as side-effect equivalence. We have shown above how to construct a universe of terms so that two schemata are side-effect equivalent iff they are equivalent over the domain of terms. It is impossible to use the universe of terms as a universal domain of interpretation when the use of replacement operators is allowed.

8.3. Current Problems and Future Work

How can we characterize more precisely the difference between functions that need to use a recursive or parallel control structure as opposed to those that only need a simple iterative program structure? The problem of deciding whether any given recursive schema can be rewritten as a program schema is of course undecidable. We would like to find general criteria of independent interest which would be sufficient to guarantee that a recursive schema could not be rewritten as a program schema.

There is general agreement that the theory of computation is currently not in good shape. The three major areas (automata theory, recursive function theory, and special case hacks) are not applicable to practical programs. We can contrast our plight with the situation in applied physics. An applied physicist finds that it is essential to understand fundamental physical laws both in designing his experiments and in interpreting their results. No such fundamental laws and principles are known in programming. Recursive function theory sets the very outer limits of what is possible. Few theories are more elegant. However, the fact that classical recursive function theory deals with the indices of the partial recursive functions and not with the meaning of the programs has been a fundamental limitiation on the applicability of the theory. For example the recursion theorem says that fixed points exist for any acceptable

Goedel numbering. Almost all the classical theorems of recursive function theory can be derived using only the Godel axioms for indices of partial recursive functions. Similarly, the complexity theory of the recursive functions can be derived from Blum's axioms for indices. Automata theorists have been able to discover some of the structure of various limited classes of automata such as finite state machines, push down machines, and space and time bounded machines. However, since the theory developed has been mostly concerned with closure and complexity properties of the special machines considered as acceptors, it has had limited applicability to real computer programs. Most programs are not structured in the way required to fall into one of the special classes of machines. Some theorists hope that by studying enough examples of very narrow domains of algorithms where we have a lot of domain dependent knowledge that we can induct a theory of computation in a Baconian fashion. Deep studies have been made on questions such as how fast integers can be multiplied and how fast matrices can be multiplied. Studies in the theory of searching and sorting appear to be more relevant for constructing a unified theory of computation since they are concerned with basic computational abilities.

Studying the properties of programs schematically offers several advantages. Schemata can be programmed in a realistic fashion. They mirror the structure of programs that are used in applications. Using them we can precisely define structural properties. Properties of the structural classes can be demonstrated. Schemata give us a tool by which we can rigorously formulate and prove statements that every programmer intuitively knows. We have used schemata to make a kind of distinction between semantic and syntactic extensions to programming languages. The intent of the restriction that functions be uninterpreted is to try to prevent our mathematics from falling into what Perlis likes to call the "Turing Machine Tar Pit." By using uninterpreted function symbols we can prove both analytic and constructive theorems about classes of programs. In the analytic theory the mathematical properties of the structural classes is expounded. In the constructive theory the process by which schemata can be constructed from goal oriented language such as PLANNER. The intention is only partially realized and we must search for other natural mathematical structures to impose on our schemata in order to obtain a more realistic theory of semantic extensions to programming languages. We are continuing to investigate what gains in efficiency can be obtained from the following extensions to programming languages:

recursion

backtrack control structure

PLANNER primitives

Locations as a type

resets

free identifiers

parallel evaluation

replacement operators for constructors.

identity test as a primitive

11. BIBLIOGRAPHY

Balzer, R. "Dataless programming" Proceedings FJCC, 1967.

Balzer, R. "EXDAMS - Extendable Debugging and Monitoring System" Proc. SJCC 1969, 34. May, 1969.

Balzer, R. M. "On the Future of Computer Program Specification and Organization" December 1970.

Black, F. 1964. "A Deductive Question Answering System" Doctoral Dissertation, Harvard University, Cambridge, Mass.

Bobrow D., Teitleman W., and Darley L. "The BBN-LISP System".

Cooper, D. C. "The Equivalence of Certain Computations" The Computer Journal, Vol. 9, no. 1.

Conway, Melvin E. "A Multiprocessor System Design" AFIPS Conference Proceedings, XXIV (Fall, 1963), 139-146.

Dahl, O., and Nygaard, K. "SIMULA - an ALGC-Based Simulation Language" CACM. Sept. 1966.

Davies, D. J. M. POPLER: A POP-2 PLANNER. MIP-89. School of A-I. University of Edinburgh.

Dennis, Jack B. "Programming Generality, Parallelism and Computer Architectue" Computation Structures Group Memo No. 32. August 1968.

Earley Jay. "Toward an Understanding of Data Structures" Computer Science Departement, University of California, Berkeley.

Evans, T. G. "A Heuristic Program to Solve Geometric-Analogy Problems" Proceedinds Spring Joint Computer Conference. 1964.

Fikes, R. "Ref-Arf: A System for Solving Problems Stated as Procedures" Artificial Intelligence (1970).

Fisher, D. A. "Control Sturctures for Programming Languages" 1970.

Floyd, R. W. "Assigning meanings to Programs" Proceedings Of Symposia in Applied Mathematics. Volume XIX.

Floyd, R. W. "Nondeterministic Algoritnms" JACM. Oct. 1967.

Green, C. C. and Raphael B. "Research on Intelligent Question-answering Systems" May 1967

Green C. C. "Application of Theorem Proving to Problem Solving" Pooc IJCAI.

Guzman, A. "Some Aspects of Pattern Recognition by Computer" M.S. thesis, Massachusetts Institute of Technology, 1967.

Guzman, A. and McIntosh, H. V. "Convert" Communications of the Association for Computing Machinery, August, 1966.

Hewitt, C. "PLANNER: a Language for Proving Theorems" Artificial Intelligence Memo 137, Massachusetts Institute of Technology (project MAC), July 1967.

Hewitt, C. "PLANNER: A Language for Manipulating Models and Proving Theorems in a Robot" Proceedings of the International Joint Conference on Artificial Intelligence. Washington D. C. May 1969.

Hewitt, .c. "Procedural Embedding of Knowledge in PLANNER". Proceedings of the Second International Joint Conference on Artificial Intelligence. London Sept. 1971.

Hewitt, C. "Description and Theoretical Analysis (using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot" Phd. Feb. 1971.

Hewitt, C. "Functional Abstraction in LISP and PLANNER" Artificial Intelligence Memo 151. January 1968. Massachusetts Institute of Technology (project MAC),

Hewitt, C. "Teaching Procedures in Humans and Robots" Conference on Structural Learning. April 5, 1970. Philadelphia, Pa. Journal of Structural Learning.

Hewitt, C. and Patterson M. "Comparative Schematology" Record of Project MAC Conference on Concurrent Systems and Parallel Computation. June 2-5, 1970. Available from ACM.

Kaplan, D. M. "Correctness of a Compiler for ALGOL-like Programs" Stanford A.I. Memo No. 48.

Kay, Alan C. "Reactive Engine" Ph. D. thesis at University of Utah, 1970.

Kellogg, C. "A Natural Language Compiler for On-line Data Management" Proc. of FJCC, 1968. pp. 473-492.

Kleene, S. C. "Introduction to Metamathematics' Van Nostrand.

Manna Z. and Waldinger R. J. "Towards Automatic Program Synthesis" July 1970.

McCarthy, J. 1959. "Programs with Common Sense, Proceeding of the Symposium on Mechanisation of Thought Processes" National Physical Laboratory, Teddington, England, London: H. M. Stationery Office, pp. 75-84.

McCarthy, J.; Abrahams, Paul W.; Edwards, Daniel J.; Hart, Timothy P.; and Levin, Michael I. "Lisp 1.5 Programmer's Manual, M. I. T. Press"

McCarthy, J. "A Basis for a Mathematical Theory of Computation" Computer Programming and Formal Systems. North-Holland, Amsterdam 1967.

McCarthy, J. "Definitions of New Data Types in ALGOL X" ALGOL Bulletin. OCT. 1964.

McCarthy, J. and Hayes, P. "Some Philosophical Problems form the Standpoint of Artificial Intelligence" Stanford A. I. Memo 73.

Minsky, Marvin. "Matter, Mind, and Models" in Semantic Information Processing. pp. 425-432.

Minksy, Marvin. (ed.) "Semantic Information Processing" M.I.T. Press, Cambridge, Mass., 1968.

Minsky, Marvin. "Form and Content in Computer Science" JACM. Jan. 1970.

Newell, A., Shaw, J. C., and Simon, H. A. "Report on a General Problem-solving Program" Proceedings of the International Conference on Information Processing, Paris: UNESCO House, pp. 256-264.

Paterson, M. S. "Equivalence Problems in a Model of Compuation" Ph. D. Thesis University of Cambridge. August 1967.

Paterson, M. S. "Program Schemata" Machine Intelligence III.

Perlis, A. J. "The Synthesis of Algorithmic Systems" JACM. Jan. 1967.

Raphael B. "The Frame Problem in Problem-solving Systems" June 1970.

Rulifson, Johns; Waldinger, Richard; and Derksen, Jan. "A Language for Writing Problem-Solving Programs" IFIP 1971.

Rovner, Paul D. "LEAP Users Manual" Lincoln Laboratory Technical Memorandum No. 231-0009.

Sandewall, E. "Formal Methods in the Design of Question-answering Systems. Uppsala University Department of Computer Sciences Report NR 28. Oct. 1970.

Slagle, J., 1965. "Experiments with a Deductive Question-answering Program" Communications of the Association for Computing Machinery, December, 8:792-798.

Strong, H. R., Jr. "Translating Recursion Equations into Flow Charts" Conference Record of Second Annual ACM Symposium on Theory of Computing.

Waldinger R., "Robot and State Variable" April 1970.

Waldinger and Lee, "PROW: A Step Toward Automatic Program Writing" Proc. IJCAI.

Winograd, T. " Procedures as a Representation for Data in a Computer Program for Understanding Natural Language" MAC TR-84. February 1971.

Woods, W. A. "Procedural Semantics for a Question-Answer Machine" Proc. FJCC. 1968. pp. 457-471.

Biographical Note

Carl Hewitt was born in Clinton, Iowa, but considers himself a native of El Paso, Texas to which he moved at the age of two years. He attended El Paso Public Schools and graduated from El Paso High School in 1963. With a Mc Dermott Scholarship, he attended M.I.T. In 1967 he graduated in mathematics, receiving a fellowship to do graduate work in artificial intelligence and theories of computation.

His publications include:

"Automata on a Two Dimensional Tape" (with Manuel Blum). Annual Conference on Switching and Automata Theory. October 1967. Austin, Texas.

"Comparative Schematology" (with Michael Paterson). Proceedings of Project MAC Conference on Parallism. June 1970. Woods Hole, Mass.

"PLANNER: A Language for Proving Theorems in obots" Proceedings of IJCAI. May 7-9, 1969. Washington D. C.

"Teaching Procedures in Humans and Robots" Proceedings of Conference on Structural Learning. April 5, 1970. Philadelphia, Pa.

```
tabs are 8 spaces
  !"#$%&'()
 *+,-./0123
 456789:;<=
 >?@ABCDEFG
 HIJKLMNOPQ
 RSTUVWXYZ[
 ¢]¬_'abcde
 fghijklmno
 pqrstuvwxy
 z{|}°
```

10. Index of Procedures

The type hierarchy is given at the beginning of chapter 4. The syntax primitives are given after the function READ. The page number gives the explanation of the procedure.

Index of Procedures

!*	188
!*	150
!+	150
!+	187
*	149
+	149
-	149
/	149
==	177
==?	134

=?	134
?	180
?	180
ABS	149
ACTIVE	179
ACTOR	79
ACTOR-CALLER	99
ACTOR-FUNCTION	77
AGAIN	111
ALL	178
ALPHABETIC	154
ALTER!-PERSISTENT	191
ALTER!-TENTATIVE	192
AND	93
AND?	92
ANTECEDENT	210
ARC	133
ARGS	171
AS	191
ASCENDING?	148
ASSERT!-PERSISTENT	215
ASSERT!-TENTATIVE	214
ASSIGNED?	171
AT	133
ATC	167

ATOM!-CONSTRUCTOR	146
ATOM!-DECOMPOSER	180
BE	178
BINDINGS	171
BITS	146
BLOCK	157
BLOCKBIND	109
BOTTOM	140
BOUND?	172
BYTE	146
CALL	111
CALL	[illegible]
CALL	120
CALL	101
CANDIDATES	213
CATCH	94
CHANNEL	154
CHARACTER	154
CHARACTER-VECTOR	86
CHOP!-PERSISTENT	139
CHOP!-TENTATIVE	139
CLAUSE	239
CLAUSE	238
CLAUSE-OF	239
CLOSE	154

CLOSURE	130
COND	94
CONSEQUENT	208
CONTAINS	183
CONTINUE	128
COPY	134
CURRENT	223
CURRENT?	219
DAGGER	181
DEBUG	170
DEC!-PERSISTENT	147
DEC!-TENTATIVE	147
DECLARATION	179
DECLARE	99
DECLARED	167
DELETE-FRONT	153
DELETE-REAR	153
DESCENDING?	148
DIGIT	154
EDIT	198
EITHER	81
ELEMENT	183
EMPTY	183
EMPTY?	130
END-BLOCK	157

ERASE!-PERSISTENT 219

ERASE!-TENTATIVE 218

ERASING 211

ERROR 170

EVAL 106

EXHAUST 235

EXPT 149

EXTENSION 86

EXTRACT 189

FAIL 112

FAIL 235

FAILPOINT 96

FIELDS 187

FINALIZE 231

FIND 233

FOLLOWS 185

FOR 113

FOR 231

FOR-RESOLVENT 240

FORK 127

FRAME 171

FRONT 152

FUNCTION 72

GATE 227

GENLOC 151

GET	132
GET!-NO-monitor	132
GETC	167
GIVEN	191
GOAL?	225
GOALS	225
GREATER	187
GREATER=	187
HAS	181
HEAD	134
HOMOGENEOUS	88
IDIVIDE	148
IN	151
INC!-PERSISTENT	147
INC!-TENTATIVE	147
INCREASING?	141
INDEFINITE	143
INDEX	140
INDICATOR	134
INITIAL	133
INSERT-FRONT	153
INSERT-REAR	153
INSTANTIATE	239
INTEND	293
INTENT	292

INTERRUPT	129
INVOKE	118
IS	105
IS-ACTOR	178
IS?	105
ISOMORPHIC?	136
ITUPLE	143
IVECTOR	143
LAST?	133
LENGTH	140
LESS	186
LESS=	186
LINEAR	183
LINK	157
LIST!-CONSTRUCTOR	142
LIST!-DECOMPOSER	185
LOCATIVE	138
LOCK	168
LOCKED?	169
LOCKER	169
LOWER	154
MACRO	78
MATCH	106
MATCH?	105
MATCHING	178

MAX	150
MEMBER?	142
MIN	150
MONAD	183
MONAD?	130
MONITOR	164
NAME	171
NEXT	133
NEXTCH	157
NODE!-CONSTRUCTOR	145
NODE!-DECOMPOSER	166
NON	177
NOT?	93
NUMBER	186
OBLIST	156
OF	132
OF-TYPE	191
ON	157
OPTIONS	181
OR	92
OR?	93
OVERALL	294
PERSIST	232
PERSISTENT	104
PNAME	146

POP 152

PORT 122

POWER 189

PRECEDES 184

PRIMARY 226

PRIN1 155

PRINC 155

PRINT 156

PROCBIND 110

PROCEDURE 171

PROCESS 119

PROCNAME 171

PRODUCT 82

PRODUCT-OF 189

PROG 106

PROTECT 163

PROTECTION 164

PUSH 152

PUT!-NO-MONITOR 138

PUT!-PERSISTENT 137

PUT!-TENTATIVE 138

PUTC!-PERSISTENT 167

PUTC!-TENTATIVE 167

PUTLOC!-PERSISTENT 151

PUTLOC!-TENTATIVE 151

PUTREST!-PERSISTENT 138

PUTREST!-TENTATIVE 138

QUOTE 106

READ 158

READCH 157

REAR 153

REPEAT 112

REPLACE 184

RESOLVE 239

REST 131

RESTORE 97

RETRACT 166

RETRY 230

RING 152

RULE 98

SELECT 182

SET-ALARM 90

SET-TIMER 90

SHARE 145

SIGNED-BITS 146

SIMILAR? 135

STACK 152

STAR 181

STATE 226

STEP 136

STOP 128

STORAGE 166

STRAIGHTEN 103

STRAIGHTEN-UP 104

STRING!-CONSTRUCTOR 145

STRING!-DECOMPOSER 185

STRUCTURE 182

STRUCTURE? 129

SUBGOAL 225

SUBSTITUTE 141

SUM-OF 188

SUPPRESS 106

SUSPEND 128

SWITCH 228

SWITCH 226

TAIL 134

TEMPORARY 101

TEMPORIZE 102

TEMPROG 227

TERMINATE 128

TOP 140

TRAILER 156

TYPE 166

TYPE-VECTOR 85

UNASSIGN 172

UNEXTEND 86

UNFALSE 91

UNIQUE 230

UNIQUELY? 141

UNIQUIZE 140

UNLOCK 170

UNMONITOR 166

UNPROTECT 163

UNSET-ALARM 90

UNSET-TIMER 90

UNSHARE 144

UPDATE 226

UPPER 154

VALUE 151

VARIABLES 239

VECTOR1-CONSTRUCTOR 145

VECTOR1-DECOMPOSER 185

VEL 177

WAIT-CALL 126

WAIT-GET 132

WHEN 179

- 105