MASSACHUSETTS INSTITUTE OF TECHNOLOGY
PROJECT MAC

Artificial Intelligence
Memo No. 168

MAC-M-386
October 1968
Revised June 1969
Revised September 1969
Revised August 1970

PLANNER:
A Language for Manipulating Models and Proving Theorems in a Robot

Carl Hewitt

Work reported herein was supported by the Artificial Intelligence Laboratory, an M.I.T. research program sponsored by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract number N0O014-70-A-0362-0002.

Reproduction of this document, in whole or in part, is permitted for any purpose of the United States Government.

SUMMARY

PLANNER is a language for proving theorems and manipulating models in a robot. The language is built out of a number of problem-solving primitives together with a hierarchical control structure. Statements can be asserted and perhaps later withdrawn as the state of the world changes. Conclusions can be drawn from these 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 make the language efficient. The use of a general-purpose matching language makes the deductive system more powerful. The language is being applied to solve problems faced by a robot and as a semantic base for English.

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
        4.1.1 Expressions
    4.2 Types
    4.3 Primitive Forms
        4.3.1 Functional Forms
        4.3.2 Macro forms
        4.3.3 Actor Forms
    4.4 Simple examples of Matching
    4.5 Actors in Patterns
        4.5.1 Primitive Actors
            4.5.1.1 Control Structure Primitives
            4.5.1.2 Data Structure Primitves
                4.5.1.2.1 Pointer
                4.5.1.2.2 Atom and Property List
                4.5.1.2.3 Word and Number
                4.5.1.2.4 List
                4.5.1.2.5 Vector and Tuple
                4.5.1.2.6 Algebraic
        4.5.2 Examples of the Use of Actors
    4.6 Functions in Expressions
        4.6.1 Primitive Functions
            4.6.1.1 Control Structure Primitives
                4.6.1.1.1 Single Process
                4.6.1.1.2 Multi-Process
            4.6.1.2 Data Structure Primitves
                4.6.1.2.1 Pointer
                4.6.1.2.2 Atom and Property List
                4.6.1.2.3 Word and Number
                4.6.1.2.4 List
                4.6.1.2.5 Vector and Tuple
                4.6.1.2.6 Algebraic
                4.6.1.2.7 Process
        4.6.2 Examples of the Use of Functions
    4.7 The Assembler MUMBLE

4.7.1 Commands
4.7.1.1 Control Structure Commands
4.7.1.2 Data Structure Commands
4.7.2 Predicates
4.7.2.1 Primitive Predicates
4.7.2.2 Compound Predicates
4.7.3 Macros
4.7.4 Examples

5. The Theorem Proving Language PLANNER
5.1 PLANNER Forms
5.1.1 Hierarchical Control Structure
5.1.2 Functional Forms
5.1.3 Theorems
5.1.3.1 Antecedent
5.1.3.2 Consequent
5.1.3.3 Erasing
5.2 Primitives
5.2.1 Data Structure Primitives
5.2.1.1 Assertions
5.2.1.2 Erasures
5.2.1.3 Goals
5.2.2 Control Structure Primitives
5.2.2.1 Failure
5.2.2.2 Finalization
5.2.2.3 Repition
5.3 Clauses in PLANNER
5.4 A Trivial Example
5.4.1 Using a Consequent Theorem
5.4.2 Using an Antecedent theorem
5.4.3 Using Resolution

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
6.1.5 Semantics of Natural Language
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 Procedural Abstraction
7.2.2.1 Protocols
7.2.2.2 Variabalization 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. Bibliography

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. This work will hopefully be part of a dissertation. 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 memo. T. Winograd and G. Sussman made suggestions for improving PLANNER.

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 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 some where 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 properties of procedures written in 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.

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 dont 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. "dont'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 dont accept the Hypothetical—"

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

"—but you dont 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 or 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

Several fundamental questions must be faced by any foundation for problem solving. 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 of problems. As part of the definition of the formalisms the following elements must be defined: the data structure, the control structure, and the primitive procedures. The problem of what are allowable data structures for facts about the world immediately arises. A foundation for problem solving must confront the problem of change: How can account be taken of the changing situation in the world? What are good ways to express problem solution methods and how can plans for the solution of problems be formulated? How can new problem solving procedures be synthesized out of goal oriented language? What properties of its procedures will a problem solver be able to know and how will they be established? All of the above questions must be addressed by a foundation for problem solving.

We shall propose a foundation for problem solving in which a formalism called PLANNER will play a central role. If it is flexible enough, the same formalism can provide the basic

facilities for a foundation for problem solving. PLANNER is a unified collection of problem solving primitives for proving theorems and manipulating models in a robot. We have tried to make this collection complete in the sense that it should be possible to define any fundamental problem solving process in terms of the primitives in a natural and elegant manner. The primitives are designed to be tied together by a HIERARCHICAL CONTROL STRUCTURE which is different from the control structure of recursive subroutine calls. Roughly speaking in hierarchical control structure the hierarchy of the previous calls is preserved so that a process can back up a previous state if it so desires. Hierarchical control makes PLANNER very convenient for constructiong elaborate hypothetical structures. It is in the above sense that we shall speak of PLANNER as a language. PLANNER is a high level, nonprocedural, goal-oriented language 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 by specifying the operations to be performed. Many of the primitives have been developed as extensions to the language 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 closeure porperties in the language. We would be grateful to any reader

who could suggest problems that would seem to require further extensions or modifications to the language. The language will be explained by giving an over-simplified picture and then attempting to correct any misapprehensions that the reader might have gathered from the rough outline.

One basic idea behind the language is a duality that we find between certain imperative and declarative sentences. For example consider the statement (implies A B). As it stands the statement is a perfectly good declarative statement. It also has certain imperative uses for PLANNER. For example it says that we should set up a procedure which will note whether A is ever asserted and if so to consider whether B should then be asserted. Furthermore it says that we should set up a procedure that will watch to see if it ever is our goal to try to deduce B and if so whether it is wise to make a subgoal to deduce A. 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 such things as universal quantifiers, conjunctions, disjunctions, etc. also have both declarative and imperative uses. PLANNER theorems are being used as imperatives when they are being executed and as declaratives when used as data.

Our work on PLANNER has been an investigation in PROCEDURAL EPISTEMOLOGY, the study of how knowledge can be embedded in procedures. The PRINCIPLE OF PROCEDURAL EMBEDDING

is that intellectual structures can be analyzed through their procedural analogues. We will show have the following all have procedural analogues:

descriptions
recommendations
theorems
proofs
grammars
models of programs
patterns

Descriptions have procedural analogues in the form of PLANNER procedures which recognize the objects described. Theorems in the predicate calculus correspond to PLANNER theorems for making deductions. Mathematical proofs correspond to plans in PLANNER for generating a valid chain of deductions. The PROGRAMMAR language of Terry Winograd provides a procedural analogue to obtain the kind of information that is supposed to be supplied by transformational grammars. Intricate patterns can be specified in procedural pattern matching languages. Models of programs are defined by procedures which state the relations that must hold between the varibles of the program as control passes through various points.

From the above observations, we have constructed a language that permits both the imperative and declarative aspects of statements to be easily manipulated. PLANNER uses a pattern-directed information retrieval system. When a statement is asserted recommendations determine what conclusions will be drawn from the assertions. 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. Procedures written in the language 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. We would like to use PLANNER to write a block control language in which we could specify how blocks can be moved around by a robot. Then we could write a structure building language 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. In several respects the deductive system is more powerful than the quantificational calculus of order omega. We have tried to design a sophisticated deductive system together with an elaborate control structure so that lengthy computations can be carried out without blowing up. Of course procedures written in PLANNER are not interinsically efficient. A great deal of thought and effort must be put inot 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. In general a uniform proof procedure gives us very little control over how or when a theorem is to be used. The problem is one of the level of the interpreter that we want to use. A digital computer by itself will only interpret the hardware instructions of the machine. We can write a higher level interpeter such as LISP that will interpret assignments and recurseive function calls. At a still higher level we can write an interpreter such as MATCHLESS which will interpret

patterns. At the level of PLANNER we can interpret assertions, find statements, and goals. It goes without saying that we can compile code 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 achived 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. Because of 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. Examples of some of the kinds of statements that can be made in the language are:

Find the second smallest integer that is sum of its factors.

Pick up all the red cubes that are on top of blue cubes and put them in the yellow box.

Assert that all the people in this room are older than Jack.

Find all the employees at MIT that are related to each other and give the relationship of each to the others.

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. Examples of structural methods are the use of subgoals in PLANNER and 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 to bring 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 the "for" statement of MATCHLESS, the "find" statement in PLANNER, and recursion on the car and the cdr in LISP. The "find" and "for" primitives are explained in the MATCHLESS and PLANNER documentation. The patterns 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 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 of recursion is appropriate in any given problem. The problem of syntesizing

programs out of canned loops is formally identical to the problem of finding proofs by mathematical induction. Indeed many proofs can be fruitfully considered to define procedures which are proved to have certain properties. 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.

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 developed in Principia Mathmatica. 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".

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 are 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 which is domain independent. 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 to try solve its problems. But surely, it is to the credit of the program that it is able to accept new 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 is used 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 general solution to the problem of finding general methods for reducing the combinatorial search involved to solve a problem using an arbitrary set of axioms. It is intended to be a general formalism in which knowledge in 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 elegance, generality, and efficiency.

3. Discursive Overview

This chapter contains an explanation of some of the ideas in PLANNER in essay form. It is based on a draft written by T. Winograd for the course 6.545. If the reader would like to see a more logically systematic presentation, he can consult the subsequent chapters. The easiest way to understand PLANNER is to watch how it works, so in this chapter, we will present a few simple examples and explain the use of some of its most elementary features.

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))
(DEFINE THEOREM1
  (CONSEQUENT (X) (FALLIBLE $?X)
    (GOAL (HUMAN $?X))))
```

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

```
(GOAL (FALLIBLE TURING))
```

We immediately see several points. First, there are two different ways of storing information. Simple assertions are stored in a data base of assertions, while more complex sentences containing quantifiers or logical connectives are expressed in the form of theorems (although facilities exist for storing them in standard logical notation and handling them as in any other theorem prover if that is desired).

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 (which is hash-coded in various ways to give the system efficient look-up capabilities). DEFINE puts a new theorem in the data base. 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 $?X), we can do this by accomplishing the goal (HUMAN $?X), where X is a variable. The strange prefix characters are part of PLANNER's pattern matching capabilities (which are extensive and make use of the pattern-matching language MATCHLESS. If we ask PLANNER to prove a goal of the form (A X), there is no obvious way of knowing whether A and X are constants (like TURING and HUMAN in the example) or variables. 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 a variable must be indicated by adding an appropriate prefix. This prefix differs according to the exact use of the variable in the pattern, but for the time being let us just accept $? as a prefix indicating a variable. The definition of the theorem indicates that it has one variable, X by the (X) following CONSEQUENT.

The third statement illustrates the function GOAL, which calls the PLANNER interpreter to try 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 variable $?X 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))
        (PROVED? (FALLIBLE TURING))
        FAIL
ENTER THEOREM1
X BECOMES TURING
        (GOAL (HUMAN TURING))
                (PROVED? (HUMAN TURING))
                SUCCEED
```

The way in which variables are bound by matching is of key importance to PLANNER . Consider the question "Is anything fallible?", or in logic (EXISTS Y (FALLIBLE Y)). This could be expressed in PLANNER as:

```
(THPROG (Y) (GOAL (FALLIBLE $?Y)))
```

Notice that THPROG (PLANNER's equivalent of a LISP PROG, complete with GO statements, tags, RETURN, etc.) in this case it acts as an existential quantifier. It provides a binding-place for the variable Y, but does not initialize it -- it leaves it in a state particularly marked as unbound. To answer the

question, we ask PLANNER to evaluate the entire THPROG expression above. To do this it starts by evaluating the GOAL expression. This searches the data base for an assertion of the form (FALLIBLE $?Y) 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 variable X in the theorem is identified with the variable Y in the goal, but since Y has no value yet, X does not receive a value. The theorem then sets up the goal (HUMAN $?X) with X as a variable. The data-base searching mechanism takes this as a command to look for any assertion which matches that pattern (i.e. an instantiation), and finds the assertion (HUMAN TURING). This causes X (and therefore Y) 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 the theorem:

```
(CONSEQUENT (X) (FALLIBLE $?X)
    (GOAL(HUMAN $?X)))
```

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

```
(THPROG (X) (GOAL (FALLIBLE $?X)) (GOAL (GREEK $?X)))
```

This time the protocol is

```
[GOAL (FALLIBLE $?X)]
        [PROVED? (FALLIBLE $?X)]
        X BECOMES TURING
                [GOAL (GREEK TURING)]
                        [PROVED? (GREEK TURING)]
                        FAIL
                FAIL
        X BECOMES SOCRATES
                [GOAL (GREEK SOCRATES)]
                        [PROVED? (GREEK SOCRATES)]
                        SUCCEED
```

THPROG acts like an AND, insisting that all of its terms are satisfied before the THPROG is happy. Notice what might happen. The first GOAL may be satisfied by the exact same deduction as before, since we have not removed information. If the data-base searcher happens to run into TURING before it finds SOCRATES, the goal (HUMAN $?X) will succeed, binding $?X to TURING. After (FALLIBLE $?X) succeeds, the THPROG 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 "push-down list" now contains

only the THPROG. 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 backup in case of failure is always possible, and moreover this backup 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. Other decisions might be the choice of a theorem to satisfy a goal, or a decision of other types found in more complex PLANNER functions. 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 $?X) 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 THPROG 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 THPROG, 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 backing up and trying things over (this is

actually the process of trying different branches down the subgoal tree.) All of the functions like THCOND, THAND, THOR, etc. are controlled by success vs. failure, rather than NIL vs. non-NIL as in LISP. This is then the PLANNER executive which establishes and manipulates subgoals in looking for a proof.

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 a subgoal tree. Other evaluators, such as LISP, with a basic recursive evaluator have no way to do this. 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 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:

```
(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, asssertion, 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 processs 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 must be 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:

```
(FORALL X Y Z S
       (AND
              (ON X Y (PUT X Y S))
              (OR
                     (EQUAL Y Z)
                     (NOT (ON X Z (PUT X Y S))))))
```

In this representation, PUT is a function whose value 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 (ON 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 (ON $?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 to make great simplifications. Now we can simply assert (ON B1 B2)

without any explicit mention of states. We can express the necessary theorem as:

```
(DEFINE THEOREM3
        (CONSEQUENT (X Y Z) (PUT X Y)
                (GOAL (ON $?X $?Z))
                (ERASE (ON $?X $?Z))
                (ASSERT (ON $?X $?Y))))
```

This says that whenever we want to satisfy a goal of the form (PUT $?X $?Y), we should first find out what thing Z the thing X is sitting on, erase the fact that it is sitting on Z, and assert that it is sitting on Y. We could also do a number of other things, such as proving that it is indeed possible to put X on Y, 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 varients 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 statemsnts in the two states.

If we look back to the distinction between assertions and theorems made on the first page, 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 0) = TRUE: 0=0 SO 1
Thus (f 0) = 1

(f 1) = FALSE: 1=0 SO
        1 * new [1-1] TRUE: 0=0 SO 1
Thus (f 1) = 1

(f 2) = FALSE: 2=0 SO
        2 * new [2-1] FALSE: 1=0 SO
                1 * new [1-1] TRUE: 0=0 SO 1
Thus (f 2) = 2

(f 3) = FALSE: 3=0 SO
        3 * new [3-1] FALSE: 2=0 SO
                2 * new [2-1] FALSE: 1=0 SO
                        1 * new [1-1] TRUE: 0=0 SO 1
Thus (f 3) = 6
```

By the process of "variabalization", 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 x) = if x=0 then 1
        else x * new [(x-1)->x] if x=0 then 1
                else x * new [(x-1)->x] if x=0 then 1
                        else x * new [(x-1)->x] if x=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 in fact 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. The teacher 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.

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 to other structural pattern matching languages such as CONVERT. It has been designed with the following considerations in mind:

1. The language should be very powerful yet simple constructs should be efficiently compiled.

2. Functions must be able to be separately compiled.

3. The syntax must be completely unambiguous as to which elements are function calls, identifiers, and data structures.

4. The language must interface with PLANNER in a natural way.

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

4.1 The Syntax of Identifiers and Expressions

4.1.1 Prefix Operators for Identifiers

As is usual in pattern matching languages we shall allow constants like 3, a, (a b), and (c (1 g)) to match only themselves. An identifier will be indicated by a prefix operator which will tell how the identifier is to be used. For example $$x is the value of the identifier x. If x has the value (a 3) then $$x will only match (a 3). We shall use $"x ["the location of x"] for the location where the value of the identifier x is stored. 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 temporary value of x"] against (a b), then x will immediately be 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. Note that model 37 teletypes obstinately convert the character "back arrow" to the character ←. Again suppose that x has the value 3. If we match $:x ["the permanent value of x"] against (a b), then x will immediately be given the value (a b). However the value of x will remain (a b) whether or not the remainder of the pattern matches. Once again suppose that x has the value 3. If we match $;x ["the subsequent value of x"] against (a b) then

the value of x will remain 3 unless the rest of the pattern matches. If the rest of the pattern matches then x will be assigned the value (a b).

4.1.2 Syntax of Expressions

Matchless uses Polish prefix notation for function calls with the actual call delimited by "(" and ")". For example (+ 2 3) evaluates to 5. If y has the value 4, then (+ $$y 1) will only match 5. Of course we use the characters "(" and ")" to delimit lists. The value of ($$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). We use the characters "[" and "]" to delimit vectors. vectors are stored in garbage collected storage. The value of [$$y (a b) $$y] is [4 (a b) 4]. Tuples are delimited by "![" and "]!". They are stored in the stack whereas the vectors are garbage collected. Otherwise vectors and tuples are indistinguishable.

4.2 Types

The simple types and their abbreviations are:

(?) for pointer for example (a b), a, and ()

(atomic) for atomic for example a, foo, (), and hello

<?> for segment for example -a b-, -3 (a a a)-, --. The list (a b c) has subsegments --, -a-, -a b-, -a b c-, -b c-, and -c-.

(FIX) for fixed point number for example 3, 4, and 999

(FLOAT) for floating point number for example 3.0, .1, and 88.3

The rest of this section should not be read until the reader understands pattern matching which is explained in section 4.3 below. The following types will not be explained here. They are included only for completeness. The complicated types and their abbreviations are:

(KAPPA type type-of-arguments) for kappa expression,

(LAMBDA type type-of-arguments) for lambda expression,

(ACT) for activation,

(ST) for state,

(BIND) for bindings,

(LOC type) for location,

(TUP type) for tuple,

(VEC type) for vector.

(HANDLER) for interrupt handler

(PROC) for process

Other types can be defined. For example the type number can be defined as follows:

(define num (kappa () (vel (fix) (float))))

Define the type (num) (ie. number) to be a type with no arguments which is the disjunction of being a fixed or floating point number. The type xpr which is an s-expression can be defined by

(define xpr (kappa () (vel (atomic) (list))))

An s-expression is () or atomic or a list.

(define list (kappa () (vel () (pair (xpr) (list)))))

A list is () or the pair of an s-expression and a list.

(define pair (type (((xpr) first)(<list> rest))
        ($=first $=rest)))

A pair is a list whose first element is an s-expression and the rest is a list. The function make will construct the appropriate structure for a type with arguments. Thus (make

(pair (a) (b c))) evaluates to ((a) b c). Also (first ((a) b c)) is (a) and (rest ((a) b c)) is (b c).

```
(define property-list (kappa () (star (atomic) (xpr))))
```

A property list is a list of even length such that the odd numbered elements are atomic. The actor star is the Kleene star of regular expressions. For example the following are property lists: (), (a (3)), and (pi 4 hello (r 3)).

```
(define complex
        (type (((num) real)((num) imaginary))
                [complex $=real $=imaginary]))
```

The type complex has two arguments real and imaginary which are numbers. The function "make" applied to a type will construct an object of that type out of lists or vectors. The function "erect" applied to a type will construct an object of that type out of tuples.

```
        (make (complex 3 4)) evaluates to [complex 3 4]
        (real (make (complex 3 4))) evaluates to 3
        (imaginary (make (complex 3 4))) evaluages to 4
        (prog  (((num) a b))
                (; This a comment. We are inside a program. The
```

identifiers a and b are declared to be of type (num) ie number)

(; in the assignment statement below the pattern (complex $-a $-b) is matched against the expression (complex 3 4).)

(assign (complex $-a $-b) (make (complex 3 4)))

a gets the value 3

b gets the value 4

(real (assign (complex (replace 7) 4) (make (complex 3 4)))) evaluates to 7

(prog (:ix) (((complex) (c (erect 1 2))))

(real $$c)) will evaluate to 1

The expression (SETLOC 1 x) will set the location 1 to the value x and return the value x.

(setloc

(real

(make (complex 3 4))

loc)

2) will evaluate to (complex 2 4)

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

```
(define instruction (type
    (((fix) opcode)
    ((fix) accumulator)
    ((fix) indirect)
    ((fix) index)
    ((:ix) address))
        (fields
                ((bits 9. 27.) $=opcode)
                ((bits 4. 23.) $=accumulator)
                ((bits 1. 22.) $=indirect)
                ((bits 4. 18.) $=index)
                ((bits 18. 0.) $=address))))
```

An instruction with opcode 254 and 4 in the accumulator field will cause the machine to halt. We can construct such an instruction with (make (instruction 254 4)) which evaluates to the fixed point number "254400000000".

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 () otherwise. Pattern matching takes place through the use of side effects to change the value of identifier to be that of the object which it is matching. The assignment statement in MATCHLESS is a varient of the primitive "is". The expressions (ASSIGN pattern expression) is well defined only if pattern matches expression. The value of the function "assign" is the value of expression.

A segment identifier is always assigned the smallest possible leftmost segment as value in matching. Below we give some examples of the values of identifiers after assignment statements have been executed. We use the character - to delimit segments. The characters (and) are used to delimit function calls.

```
(prog (a ((atomic) h) (<?> c))
        (; This is a comment.  We are inside a program
in which we have declared a to be a pointer, h to be atomic, and
c to be a segment)
        (; in the assignment statement below the pattern
```

($←a k $←h $←c) is matched against the value ((l) k b l a).

(is ($←a k $←h $←c) ((l) k b o a))

a gets the value (l)

h gets the value b

c gets the value -o a-

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

(prog ((<?> c) ((atomic) h) a)

(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) (a))) fails because there is only one element in (a).

(prog (((atomic) a))

(is $←a (b t))) fails because (b 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 $$a) (a b c a o c))
        a gets the value -a b c-
(prog ((<?> a b))
        (is ($-a x $$a $-b) (a b x d x a b x o q))
        a gets the value -a b x o-
        b gets the value -q-
```

An expression that consists of the prefix operator $? (read the value given) followed by an identifier will match the value of the identifier if it has one, otherwise the identifier is assigned a value. We shall use the pseudo atom NOVALUE to indicate that an identifier does not have a value.

```
(prog (a)
        (is $?a t))
        a gets the value t
(prog (((fix) (a 5)))
        (is $?a 4))
        a is 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. If a pattern in an assignment statement cannot match the value of the second argument of the assignment statement then the assignment statement returns the value (), otherwise the value t.

4.4 Primitive forms

4.4.1 Functional Forms

Functional forms begin with the atom lambda or the atom bindlambda. They look like:

(LAMBDA type list-of-parameters expressions) where type is the type of value returned and the value of the form is the last expression in expressions. If the list of parameters is atomic then that atom is the only parameter and it is bound to the list of unevaluated arguments. Functional forms that begin with the atom bindlambda have the form (BINDLAMBDA type (list-of-parameters binding-identifier) expressions). For example

((lambda x $$x) a b c) evaluates to (a b c)

If the first element of the list of parameters is a type then there is just one argument which is a tuple of that type.

((lambda ((tup (fix)) x) (1 $$x)) 11 21 33) evaluates to 11 where (tup (fix)) is the type of the tuple of fixed point numbers. The function 1 selects the first element of the tuple.

Otherwise we will have a list of declarations of identifiers.

((lambda (x) $$x) (" 3)) evaluates to a pointer to 3 The function " is quote. Note that the value is not the fixed point number 3.

((lambda (x) $$x) a) evaluates to a

((lambda (fix) (((fix) x)) $$x) (+ 2 2)) evaluates to 4

((lambda (fix) (((fix) x)) (+ $$x 1)) 2) evaluates to 3

((lambda (fix) (((fix) x) ((fix) y)) (+ $$x $$y)) 2 3) evaluates to 5

If an identifier is of type (") then the corresponding argument is not evaluated.

((lambda ((") x)) $$x) 3) evaluates to a pointer to 3

((lambda ((") x)) $$x) a) evaluates to a

((lambda ((") x)) $$x) (+ 2 2)) evaluates to (+ 2 2)

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

```
(define + (lambda (fix) ((tup (fix)) x)
        (; the type (tup (fix)) is a tuple of fixed point
numbers)
        (for (fix)
                (((fix) (result 0))
                (; declare the identifier result to be fixed
point number and initialize it to 0))
                        ((test
                                (is ![]! $$x)
                                (return $$result)
                        (; each time before executing the loop
test to see if x is the null tuple and if so then return the
result))
                        (step (assign $!x (rest $$x)))
                        (; alter each pass throught the loop
assign x to the rest of x))
                (assign $!result (plus (1 $$x) $$result)))
                (; the body of the loop is to add the first
element of x into the result)))
```

(+ 3 2 4) evaluates to 9

4.4.2 Macro Forms

Macros are expanded by the interpreter, by the assembler, and by the compiler. The results are respectively interpreted, assembled, and compiled. Macro forms look like

(MACRO list-of-parameters expressions) The expansion of the macro is the value of the last expression. The function genbraces will generate a pair of braces. For example (1 (genbraces a b) 6) will evaluate to (1 (a b) 6).

```
(define chop (macro (x)
        (genbraces setloc
                $$x
                (genbraces rest (genbraces content $$x)))))
```

The macro chop 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.

(chop $"y) will expand to (setloc $"y (rest (content $"y))

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

```
(define + (macro x
        (cond
                ((is () (rest $$x))
                (; if the rest of x is () then the answer is 0)
                0)
                (t
```

```
                (; otherwise we want to expand to the sum of the
second element of x and "+" of the rest of the rest of x)
                (genbraces plus (2 $$x) (genbraces + <rest $$x
2>))))))
```

Thus

(+ 3 2 4) expands to (plus 3 (plus 2 (plus 4 0)))

4.4.3 Actor Forms

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.

Actor forms begin with the atom kappa or the atom bindkappa. They look like:

(KAPPA type list-of-parameters patterns)

(BINDKAPPAtype (list-of-parameters binding-identifier) patterns) where type is the type of value matched and the form must match all of the patterns. If the list of parameters is atomic then that atom is the only parameter and it is bound to the list of unevaluated arguments. For example

((kappa x $$x) a b c) matches only (a b c)

If the first element of the list of parameters is a type then there is just one argument which is a tuple of that type.

((kappa ((tup (fix)) x) $$x) 1 2 3) matches only ![1 2 3]! where (fixs) is the type of the tuple of fixed point numbers.

Otherwise we will have a list of declarations of identifiers.

((kappa (x) $$x) (" 3)) matches only a pointer to 3

((kappa (x) $$x) a) matches only a

((kappa (fix) (((fix) x)) $$x) (+ 2 2)) matches only 4

((kappa (fix) (((fix) x)) (+ $$x 1)) 2) matches only 3

((kappa (fix) (((fix) x) ((fix) y)) (+ $$x $$y)) 2 3) matches only 5

If an identifier is of type (") then the corresponding argument is not evaluated.

((kappa ((") x)) $$x) 3) matches only a pointer to 3

((kappa ((") x)) $$x) a) matches only a

((kappa ((") x)) $$x) (+ 2 2)) matches only (+ 2 2)

4.5 Actors in Patterns

Examples of actors are vel for disjunction, non for negation, et 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 to be pointers. In the assignment statement below the pattern (k <et $←a $←b> $←c) will be matched against (k x y z). The pattern <et $←a $←b> will match an expression only if both $←a and $←b match the expression.)

```
                (is (k <et $←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 <vel (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 if actor is a list of clauses. If the object that the actor is is trying to match has the property the 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 (if ((fix) $←x)) 3))
        x gets the value 3 since 3 is a fixed point
```

number.

The argument of the actor when is a list of clauses. If the first element of one of the clauses evaluates to true then the object that the actor is trying to match must match the rest of the elements in the clause.

```
(prog ((y t))
        (is (when ($$y $←y)) (a b)))
        y gets the value (a b) since it was initialized
```

to t

A number of actors are defined below. For example " is quote. Thus (" $$a) will only match $$a. 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
        (kappa ()
                (; palindrome is a actor of no arguments)
                (vel
```

```
                        ()
                        (; a palindrome is either () or)
                        (same (x)
                                (; let x be a pointer to the
first element of the list.  Also x must be the last element of
the list with a palindrome in between)
                                ($←x <palindrome> $$x)))))
```

For example

(is (palindrome) (a ! ! a)) is true.

The form kappa is like the lambda of LISP except that it is used in actors instead of in functions. The above definition reads "a palindrome is a list such that it is () or it is a list which begins and ends with x with a palindrome in between." The actor same causes the identifier x to be rebound to the pseudo-atom NOVALUE every time that palindrome is called. The actor reverse is defined to be such that (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
        (kappa (x)
                (if
                        ((atomic)
                        (; if the object being matched is atomic
then it must be equal to x)
                        $$x)
                        ((same (first (<?> rest))
                                (; otherwise let first be a
pointer to the first element of the matching object and rest be
the segment of the rest of the elements of the matching object.)
                                ($←first $←rest)
                                (; when (<reverse (rest)>
$$first) matches $$x we are done)
                                (when ((is
                                        (<reverse ($$rest)>
$$first)
                                        $$x)))))))
```

For example

(is (reverse (x y z)) (z y x)) is true

Essentially all the ideas for the actors come from Post productions, BNF, general regular expressions, SLIST, SNOBOL, CONVERT, and LISP. We will use OBJ to designate the expression that is to be used to match the actor calls being defined below. We will give examples of the use of these actors afterward. The actor braces is defined to match a pair of braces. Thus (braces 1 2) will match (1 2). The function gengraces will generate a pair of braces. The expression (genbraces 1 2) will evaluate to (1 2).

4.5.1 Primitive Actors

4.5.1.1 Control Primitves

(NON pattern) will match an object only if pattern does not match the object. Thus (non c) will match a, but (non a) will not match a.

(VEL disjuncts) where disjuncts is of type segment will match an object only if some disjunct in turn will match the object.

(ET conjuncts) where conjuncts is of type segment will match an object only if each conjunct in turn will match the object.

(IF type clauses) where clauses is of type segment will match an object if the first element of some clause in turn will match the object and then the rest of the elements in that clause match the object.

```
(prog (x y)
      (is
            (if ((num) $←x) ($←y))
            foo))
```

y gets the value foo since foo is not a number

(SAME type declaration patterns) where patterns has the type segment will match an object only if each pattern in turn matches the object. The actor same is like the actor et except that it declares its type and can declare identifiers.

```
(assign (same (x) $←x) 4)
```

x gets the value 4

(PATPROG type declaration body) where body is of type segment is just like the function prog which is described below except that instead of exiting by calling the function return it exits by calling the function true? or the function false which are described immediately below.

(TRUE? pattern expression bindings) makes sense only when evaluated within the actor patprog. If pattern matches

expression and the rest the match in which the patprog appears succeeds then the patprog is exited. Otherwise, the value of the function true? is a message created by the failure to match.

(FALSE message) causes the generation of a failure with a message the value of message.

(WHEN type clauses) where clauses is of type segment will evaluate the first element of each clause. If the evaluation is non null then the rest of the elements in the clause must match the object that the actor WHEN is attempting to match.

```
(prog (((fix) (y 1)) x)
      (is  (when ((is $←x (+ $$y 1)) (+ $$x
2))) 4))
```

x gets the value 2

($$ theta) will only match an expression which matches the identifier theta.

($? theta) will act like ($$ theta) if the identifier theta has a value. Otherwise ($? theta) will match an object x only if the identifier theta will match x.

($: theta) will match any expression x which will match the identifier theta and will give theta the value x.

($← theta) will match any expression x which can match the identifier theta. The identifier theta will be given the value x except that if the pattern fails to match then x will be restored to its previous value.

($; theta) will math any expression x which can match the identifier theta. If the entire pattern matches then theta will be assigned the value x.

4.5.1.2 Data Structure Primitives

4.5.1.2.1 Pointer Actors

(?) will match any s-expression.

(? n) will match 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) will only match an object if the object is equal to x. This is just another description of the actor " which is quote. For example (" $$a) will only match $$a and (" a) will only match a.

(STAR patterns) will match an object only if the object consists of a sequence (including the null sequence) of elements that match patterns. For example (star 3) will match (3 3 3) and (a <star b c> e) will match (a (b c) (b c) e).

(DAGGER patterns) will mathc an object only if the object consists of at least one sequence of elements that match

patterns. For example (dagger (fix)) will match (3 y).

(OPTIONS sequence-of-patterns) will match a sequence of elements which match a subsequence of the sequence-of-options from left to right. For example (options a (fix) (atomic)) will match (a 3).

(WRITE arg) will match any object. As a side effect it prints the value of arg.

(== x) will match an object only if the value of x is eq to the object.

(CONTAINS pat) will match any object that contains the pattern pat.

```
(define contains (bindkappa ((("") y)) b)
                 (; the identifier b is bound to the
bindings that were in force before the function contains was
entered)
                 (; the function contains takes one
argument which is not evaluated)
                 (; we must make y into an actor with no
arguments)
                 (; the function val will evaluate any
expression relative to a set of bindings)
                 (; the function genbraces will surround
its arguments with braces)
                 (container (val (genbraces actor (kappa
() $$y)) $$b))))

(define container (kappa (((kappa ()) x))
                 (; CONTAINS is an actor with one
argument x which is a kappa expression which matches pointers
and takes no arguments)
                 (if
                          (($$x)
                          (; if the actor matches the
matching object then we are done))
                          ((atomic)
                          (; if the matching object is
atomic then fail)
```

```
                                    (false))
                                    (((container $$x) <?>)
                                    (; if the first element in the
matching object contains x  then we are done))
                                    ((; else the rest of the
matching object must contain x)
                                    ((?) <container $$x>))))))
```

(SMASH location) will match an object only if location is a type that can hold a type of the the type of OBJ. If the rest of the match succeeds then the location is smashed to hold the object.

```
(prog (((fix) (a 1)))
        (is (smash $"a) 2))
        a gets the value 2
```

(REPLACE x) will match any object. If the entire assignment statement succeeds then OBJ will be replaced with x.

```
(prog (y)
        (is (let $←y ((replace a) <replace (b)>))
(c d e)))
y gets the value (a b)
```

(CONTENT pat) will match an location whose content matches the pattern $$pat.

```
(prog (((fix) (a 3) b)
        (assign (content $←b) $"a))
    b gets the value 3
```

(LOCATION pat) will match an expression whose location matches the pattern $$pat.

```
(prog
```

((\loc \?)) y))

\is [\?) \location $←y) <?>] [a b c)/

\return \in $$y))) will evaluate to c

\GENLOC x) will generate a new location on the stack holding the location of x.

\in \genloc 3)) will evaluate to 3

4.5.1.2.2 Atom and Property Actors

\ATOMIC) will only match an atom.

\is \atomic) a) is true.

\HAS properties) will match any atom with the appropriate properties where properties is of the form \indicator1 value1) \indicator2 value2)... The absence of an indicator on a property list is exactly equivalent to that indicator being present with with the null value. The actor "has" allows MATCHLESS to do pattern matching on arbitrary graph structures. Atoms represent the nodes of a graph; their properties represent the links between nodes. 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.

\WITH pattern properties) will match an expression such that its first element matches pattern and the result of getting the value of indicator will match value. The actor with is used

in PLANNER to match expressions that might have property lists.

(GIVE properties) will match any list or atom where properties is of the form ...(indicator value).... If the entire assignment statement in which the actor succeeds then the value of value1 will be stored under the value of indicator1.

4.5.1.2.3 Word and Number Actors

(NUM) will match OBJ only if OBJ is a a number. For example (num) will match 3.

(LESS n) will match any number less than the value of n.

(GREATER n) will match any number greater than the value of n.

(FIELDS specifications) where specifications is a segment will match any fixed point number which meet each specification of a field in turn. A fixed point number x will meet a specification of the form (bits pattern) only if the number which is the byte of x defined by bits matches pattern.

(fields ((bits 3. 0.) 4) ((bits 1. 35.) 1)) will match a fixed point number whose lower 3. bits are 4 and whose sign bit is on.

4.5.1.2.4 List Actors

An expression delimited by "(" and ")" will only match a list.

4.5.1.2.5 Vector and Tuple Actors

An expression delimited by "[" and "]" will only match a vector or a tuple.

4.5.1.2.6 Algebraic Actors

(SUM terms rest-of-terms) where terms is of type segment will match a sum of terms such that the rest of the terms match the pattern rest-of-terms.

```
(is (sum a b (?)) (+ c b a)) is true.

(prog (y z x)
        (is (sum (et (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 (sum $←y b $←x) (+ 1 c b d)))
y gets the value 1

x gets the value (+ d c)
```

(SUM-OF term sum) will match any sum of terms that match the pattern term such that the sum of such terms matches the pattern sum.

```
(prog (y)
```

```
                (is (sum-of (product x (?)) $-y) (+ (* 3
x) (* x a))))
        y gets the value (+ (* x a) (* 3 x))
```

(PRODUCT factors rest-of-factors) where factors is of type segment will match a product of factors such that each factor matches a pattern in factors and the rest of the factors match the pattern rest-of-factors.

```
        (is (product 5 b c) (* c b 5)) is true.
        (prog (x y)
                (is (product (at (num) $-x)  $-y (?)) (*
(+ 2 a) 3 a)))
        x gets the value 3
        y gets the value (+ 2 a)
        (prog (x)
                (is (product 3 $-x 1)  0))
        x gets the value 0
```

(PRODUCT-OF factor product) will match any product of factors that match the pattern factor such that the product of such factors matches the pattern product.

```
        (prog (x)
                (is (product-of (non (num)) $-x) (* a 3
b 5.0)))
        x gets the value (* b a)
```

(POWER base exponent) will match an exponential.

```
        (prog (x y)
                (is (power $-x $-y) (** 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

(prog (x y)
        (is (power $←x $←y) 1))
x gets the value 1

(prog (x y)
        (is (power (let (non 1) $←x) $←y) 1))
y gets the value 0
```

(COMMON factor terms rest-of-terms) where terms is or type segment will match a sum where a factor that matches the pattern factor appears in several terms.

(is (sum <common x (sum 3 a) (?)> y) (+ (* a x) y (* 3 x)) is true.

```
(define quadratic
  (kappa
        (x ((kappa ()) a) ((kappa ()) b) ((kappa ()) c))
        (sum
                <common
                        (power $$x 2)
                        (let (non 0) (non (contains $$x)) ($$a))>
                <common
                        $$x
                        (let (non (contains $$x)) ($$b))>
                (let (non (contains $$x)) ($$c))))))
```

Thus if

```
(prog (((?) special)a1 b1 c1))
        (is
                (quadratic
                        y
                        (factor (kappa () $←a1))
                        (factor (kappa () $←b1))
                        (factor (kappa () $←c1)))
                (+ a (* 3 y) (* z (** y 2) 4) (* c y))))
```

then

a1 gets the value (* 4 z)

b1 gets the value (+ c 3)

c1 gets the value a

4.5.2 Examples of the use of Actors

The rest of our examples of the use of actors will come from giving a rigorous definition of the syntax of LISP in MATCHLESS. Those readers who are not interested in the details need not read section 4.5.2 The following grammmar 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 (kappa ()
    (same
        ((<?> special) (tags ()) (boundvars ()))
        (lambda (varlist) (form)))))
```

Thus for example (top-function) will match (lambda () ()). The actor top-function introduces the pattern identifiers tags and boundvars and binds them to - - which is the null segment.

```
 (define varlist (kappa ()
        (star
            (same
                ((atomic) curvar) ((<?> special free)
boundvars))
                s-curvar
```

```
                (when ((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 boundvars.

```
(define
    iform (kappa ()
    (i1
        ((atomic) (vel (constant) (var)))
        (
            ((atomic) <?>)
            (if
                ((prog <?>) (progform))
                ((cond <?>) (condform))
                ((setq <?>) ((?) (var) (iorm)))
                ((go <?>) (goform))
                (((has ((subr (?)))) <?>) ((?) <star (form)>))
                (((has ((expr (?)))) <?>) (expr iorm))
                (((has ((fexpr (?)))) <?>) (?))
                (((has ((isubr (?)))) <?>) (?))
                (((has ((lsubr (?)))) <?>) ((?) <star (form)>))
                (((has ((lexpr (?)))) <?>) ((?) <star (form)>))
                (
                    (?)
                    (same (expr) $-expr (write ($$expr
undefined)))))))
        (((lambda <?>) <?>)) (lambda-junction))
        ((?) ((form) <star (form)>)))))
```

The above definition says that i1 a form is an atom then it must be a constant or an identifier; i1 its first element is an atom then i1 it begins with the atom prog, then it must be a progform etc.; i1 it begins with "((lambda" then it must be a lambda-function; otherwise it must be a iorm followed by a formlist.

```
(define constant (kappa (atomic) () (vel t () (num))))
```

The only constants are t, (), and numbers.

```
(define
   var (kappa (atomic) ()
   (same
       (((atomic) curvar) ((<?> special free) boundvars))
       $←curvar
       (vel
           (when ((is (<?> $$curvar <?>) ($$boundvars))))
           (write ($$curvar unbound))))))
```

An identifier is either in boundvars or it is unbound.

```
(define condform (kappa () (cond <star (<star (form)>)>)))
(define
   progform (kappa ()
   (same
       (
           ((<?> special free) tags boundvars)
          ((<?> special)
               (tags ($$tags))
               (localtags ())
               (boundvars ($$boundvars))))
       (prog
               (varlist)
           <et
               (collect-tags)
               (when ((is ($←tags) ($$localtags $$tags))))
               (star (or (atomic) (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 (kappa ()
(star
     (vel
```

```
            (same (((atomic) curtag) ((<?> special)
localtags))
                        $←curtag
                      (when ((is (<?> $$curtag <?>)
($$localtags))
                            (write (multiple tag $$curtag)))
                          (($←localtags) ($$curtag
$$localtags)))
                (?)))))

 (define
    exprform (kappa ()
    (same
        (args lambdavar)
            (
                (has ((expr (lambda $←lambdavar <?>))))
                <et (star (form)) $←args>)
            (when ((is (same-num) ($$lambdavar $$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 lambdavar are bound to the pseudo atom NOVALUE.

```
 (define
    same-num (kappa ()
    (vel
        (() ())
        (same
            ((<?> a b))
                ((?) $←a) ((?) $←b))
                (when ((is (same-num) (($$a) ($$b)))))))))
```

The pattern (same-num) will match any list with two elements provided that the elements both have the same number of elements. For example it will match any of the following lists: (()()), ((a)(b)), ((1 2 3) (3 2 1)).

```
(define
   goform (kappa ()
   (go
       (if
               ((atomic)
               (same (curtag ((<?> special free) tags))
                       $←curtag
                       (or
                               (assign (<?> $$curtag <?>)
($$tags))
                               (write ($$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
   lambda-function (kappa ()
   (same
       (args lambdavar)
           (
               (same
                   (
                       ((<?> special free) boundvars))
                       ((<?> special) (boundvars
($$boundvars))))
                   (lambda
                       (et (varlist) $←lambdavar)
                       (form)))
               <et (star (form)) $←args>)
           (when ((is (same-num) ($$lambdavar $$args)))))))
```

In a lambda-function the bound identifiers of the lambda must be added to boundvars and the lambda-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.

4.6 Primitive Functions

4.6.1 Primitive Functions

Examples of the values of various expressions are given below:

a evaluates to a

(a b c) evaluates to (a b c)

(" $$a) evaluates to $$a

(+ 1 2) evaluates to 3

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

(a b <" (a b)>) evaluates to (a b a b)

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

4.6.1.1 Control Primitives

4.6.1.1.1 Single Process

(IS pattern expression) is true only if pattern matches the value of expression.

(ASSIGN pattern expression) is defined only if pattern matches expression. The value of the function "assign" is the value of expression.

(PROG type declaration body) where the identifier body is of type segment. 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.

(; COMMENT) is a comment which will not be seen by the interpreter or the compiler.

(GO tag) will transfer control to the place defined by tag.

(GO activation) will restore the activation and continue processing from there. A declared tag is a local identifier which has as its value the activation defined by the tag. If a tag is declared, it must be declared in the block head of the block in which it is a tag. The following expression will read in the currently open file and return the elements of the file in reverse order as its value. The function my-read transfers to its argument when it reaches the end of file.

```
(prog ((<?> (value ()))(act) end-of-file))
      (;  the segment identifier value is
initialized to the null segment and the activation end-of-file
is defined by the tag below)
 again
      (assign ($:value) ((my-read $$end-of-
file) $$value)) (; if there is another expresson in the file
then put in value else go to end-of-file)
      (go again)
 end-of-file
      (return ($$value)))
```

(RETURN x) will leave the current block with the value x.

(EXIT x) will leave the current function with the value x.

(CATCH x p) will attempt to evaluate x. If an error occurs in the evaluation then the value of the function catch is the value of p. Otherwise the value of the function catch is the value of x.

(CASE type n OF expressions) where the identifier expressions is of type segment will return the value of the nth expression.

(case 2 of a b c) evaluates to b

(RULE type FOR x clauses) will give a rule for the expression x. The value of x will be matched against the first element 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 the value of x. Otherwise the value is the value of the last element of the clause.

(rule for b ((=fix))) evaluates to b

(prog (x) (rule for a ($←x ($$x $$x))))

evaluates to (a a)

(rule for c (d e)) evaluates to ()

(WHILE predicate expressions) will execute the expressions while the predicate evaluates to true. It is equivalent to the following:

```
(PROG ()
AGAIN
        (COND
```

```
                    ((NOT predicate)
                    (RETURN ())))
               expressions
               (GO AGAIN))
```

(UNTIL predicate expressions) will execute the expressions until the predicate becomes true. It is equivalent to the following:

```
(PROG ()
AGAIN
        (COND
             (predicate
             (RETURN ())))
        expressions
        (GO AGAIN))
```

```
(FOR type declaration
        ((INITIAL initial)
        (STEP step)
        (TEST predicate test-action))
body)
```

where the identifiers initial, step, test-action, and body are of type segment is defined to be an abbreviation for the following where LOOP is a unique generated label:

```
(PROG type declaration
        intial
LOOP
        (COND (predicate test-action (RETURN ())))
        body
        step
        (GO LOOP))
```

Alternatively, we have

```
(FOR type declaration
        ((INITIAL initial)
        (TEST predicate test-action)
        (LIST item condition)
```

```
        (STEP step))
        body)
```

where the identifiers initial, step, and body are of type <?> is like the for loop previously described except that the value of the for statement is the list of all the items such that $$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.

```
(PROG type (declaration ((xpr) (COLLECTED ())))
        (; declare COLLECTED to be an s-
expression initialized to nil)
        initial
AGAIN
        (COND
                (predicate
                test-action
                (RETURN $$COLLECTED)))
        body
        (COND
                (condition
                (; add item onto the end of
COLLECTED if condition is met  The function "identity" is the
identity function)
                (ASSIGN $:COLLECTED (<$$
COLLECTED> item))))
        step
        (GO AGAIN))
```

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

($$ theta) is the value of the identifier which is the value of theta.

```
(prog ((((fix) common) (x 1)) (y x))
        ($$ $$y)) evaluates to 1
```

4.6.1.1.2 Multi-Process

Often it is convenient and more efficient 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: Destroying processes

(CREATE x) will create a new process which will begin execution with the function call x. The value of the function create is the name name of the created process.

(PARALLEL name x) will start the named process with the execution of the function call x. The calling process will continue to execute the other arguments of the function in which the call to parallel is made. The value of the function parallel is the value of x. The named process will be destroyed when it returns and the returned value will be used as an argument. The delimiters |(and)| can be used to delimit pararalles calls for elements and |< and >| can be used for segments. Thus in the following expression the sum of 3 9 is computed at the same time as the sum of 7 and 9:

(h |(+ 3 9)| (+ 7 9))

(RESUME name x) will resume execution of the named process from the point that control last left it and suspend execution of the calling process. The value of x is made the value of the call to the function resume which caused control to leave the named process. Expressions of the form (RESUME name) are used to start new processes. For example (resume (create (foo 2 a))) will cause (foo 2 a) to be executed in a new process.

(FORK name x) will resume execution of the named process from the point that control last left it and continue execution of the calling process. The value of x is made the value the value of the call to the function resume which caused control to leave the named process. The value of the function fork is the value of x. Expressions of the form (FORK name) are used to start new processes which will run in parallel with the calling process. For example (fork (create (foo (bar) a))) will cause (foo (bar) a) to be executed in a new process in parallel with the calling process.

(DIE name) will cause the named process to be destroyed and return the name of the process killed. If no argument is given the process which executes the call will die.

(PASSON name x) will RESUME (see above) the named process with the value of x. The process which calls the function passon will then die.

(LOCK locations) where locations is of type segment will

attempt to lock the locations which are arguments. The process which calls the function lock will be suspended until all the locations are locked.

(LOCKED? locations) where locations is of type segment will attempt to lock the locations which are arguments. If the locations cannont be locked then the function locked? will return ().

(UNLOCK locations) where locations is of type segment will unlock the locations.

4.6.1.2 Data Structure Primitives

4.6.1.2.1 Pointer Functions

(TYPE expression) will return the declared type of expression. The function type is useful in macros to decide how to expand the macro.

(KIND identifier) will return the kind of identifier that identifier has been declared to be. The kinds of identifiers are local, special, and common.

(EQ x y) is true only if the value of x is identical to the value of y.

(FUNCTION lambda-expression) will return the functional argument of lambda-expression.

(((function (lambda () 3))) evaluates to 3

(ACTOR kappa-expression) will return the actable argument of $$kappa-expression.

(is ((actor (kappa () 4))) 4) is true

(AT n x) will return the location of the the nth element of x.

(assign [<? 1> (replace a) <?>] [aa bb cc]) is equivalent to (setloc (at 2 [aa bb cc]) a)

(IN location) will return the contents of $$location as its value.

(prog (fix) (((fix) (x 1))) (in $"x)) will evalutate to 1

(SETLOC location value) will store the value in the location and return the value.

(prog (fix) (((fix) x)) (setloc $"x 1)) will assign x the value 1

(NTH n expression) will return as its value the nth element of the value of expression which must be a list, a vector, or a tuple. (NTH n expression) may be abbreviated as (n expression).

(3 (a b c)) evaluates to c

(2 [a (b c) d]) evaluates to (b c).

(REAR n expression) will return as its value the nth element of the value of expression from the rear.

(rear 2 (a b c)) evaluates to b

(REST x n) will return as its value the result of taking the rest of x n times.

(rest (a 4 d f) 2) evaluates to (d f)

(BUTLAST x n) will return as its value the result of taking the butlast of x n times.

(butlast (a 4 d f) 2) evaluates to (a 4)

(INITIAL n expression) will return as its value the initial n elements of the expression.

(initial 3 (a b c d)) will evaluate to (a b c)

(TERMINAL n expression) will return as its value the terminal n elements of the expression.

(terminal 3 (a b c d)) evaluates to (b c d)

(LENGTH x) will return the length of the value of x.

(length (a b c)) evaluates to 3

(PRINT x) will print the pointer x.

(SUBSTITUTE x pattern z) will substitute the value of x for all expressions in z that match pattern.

(substitute a (atomic) (1 (x z))) will evaluate to (a (a a))

```
(define substitute (bindlambda ((x ((*) p) z) b)
        (subst x (val (genbraces actor (kappa ()
$$p)) $$b) z)))

(define subst (lambda (x ((kappa () (?)) p) z)
        (cond
                ((is ($$p) $$z)
                $$x)
                ((is (atomic) $$z)
                $$z)
```

```
                                    (t
                                    ((subst $$x $$p (1 $$z))
                                     <subst $$x $$p (rest $$z)>))))))
```

4.6.1.2.2 Atom and Property List Functions

(GET atom indicator) will return the value under indicator for atom if such exists. Otherwise it returns ().

(PUT atom indicator value) will put value under indicator for atom.

4.6.1.2.3 Word and Number Functions

(FIXCONS n) will return a pointer to a fixed point number equal to n.

(fixcons 9) evaluates to a pointer to 9

(FLOATCONS n) will return a pointer a floating point number equal to n.

(floatcons 9.0) evaluates to a pointer to 9.0

(FIXIN p) will return a fixed point number equal to the number pointed to by p.

(fixin (" 9)) evaluates to the fixed point number 9

(FLOATIN p) will return a floating point number equal to the number pointedto by p.

(floatin (" 9.0)) evaluatesto the floating point number 1.0

(BITS s p) will define a field of s bits that is p bits from the right end of the word.

(LOAD bits i) will return an integer which is the byte of i which is defined by bits.

(load (bits 1. 35.) -1) will load the sign bit of -1 which is 1

(DEPOSIT bits source destination) will deposit the source in the byte of the destination defined by bits and return the modified destination as its value.

(deposit (bits 1 0) 4) evaluates to 5 which is 4 with the low order bit turned on.

(FIXPRINT x) will print the fixed point number x.

(FLOATPRINT x) will print the floating point number x.

4.6.1.2.4 List Functions

Any expression enclosed within "(" and ")" will evaluate to be a list.

4.6.1.2.5 Vector and Tuple Functions

Any expression enclosed within "[" and "]" will evaluate to be a vector. On the the other hand, n expression enclosed between "[" and "]" will evaluate to be a tuple. The only difference beteen vectors and tuples is that tuples are strored

in the stack while vectors are garbage collected.

(VECTOR n template fcn) will create a vector of length the value of n with entry i initialized to (fcn i). The expression template specifies the mark function and the print function of the vector.

```
(vector
        3
        (template
                (function (lambda ()))
                (function (lambda (($$ix) i) x)
                        (fixcons (at $$i $$x))))
        (function (lambda ($$ix) (($$ix) i))
$$i)))
```

will evaluate to (1 2 3). Given a location l as its argument the function fixcons will return a pointer to a fixed point number which has the same value as the contents of l.

(TUPLE n template fcn) will create 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 template declaration
        ((INITIAL initial)
        (TEST test final-action)
        (ADD element condition)
        (STEP step))
        body)
```

will create an indefinite tuple with template 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 the declaration.

```
(indefinite
        (fixs) (; the tuple is of type "fixs"
which is a tuple of fixed point numbers)
                ((fix) (i 1))) (; declare i to
be a fixed point number initialized to 1)
        ((test (is $$n $$i)) (; if n is equal to
i, then we have created our tuple)
        (add $$i) (; each time through the loop
add the value of i to the tuple)
        (step (assign $:i (+ $$i 1)) (; after
executing the body of the loop, increase i by 1)))
        (; the body of the loop is empty))
```

will evaluate to

```
        ;[1 2 3 4]; if the identifier n has the
```

value 4

(UNSHARE x) will create a copy of the value of x at the top level. The value of the function unshare will be equal to its argument but it will not be eq.

```
(unshare [1 x (y 2.0)]) will evaluate to [1 x (y
```

2.0)]

```
(prog ((vec) (x [a (4)])))
        (eq (2 $$x) (2 (unshare $$x))))
```

evaluates to true.

4.6.1.2.6 Algebraic

(ADD terms) where terms is of type segment will produce a sum of the algebraically simplified terms.

(add (* (** x 2) 3) 3 (* 2 x) (* 4 x) 4 (** x 2)) evaluates to

(+ 7 (* 6 x)(* 5 (** x 2))

(MULTIPLY factors) where factors is of type segment will multiply together the algebraically simplified factors.

(multiply 3 (+ x 2) (+ x -2) x) will evaluate to

(+ (* 3 (** x 3)) (* -12 x)) 4.6.2 Examples of the Use of Functions

The function factorial is defined below in order to illustrate the syntax of functions that produce values. The program should almost be self explanatory to any LISP programmer. On entrance to prog, temp is immediately bound to 1.

```
(define factorial
        (lambda  (fix) ((((fix) n))) (prog (fix) (((fix) (temp
1)))
        again   (cond ((is (less 2) $$n)
                       (exit $$temp)))
                (assign $$temp (* $$n $$temp))
                (assign $$n (- $$n 1))
                (go again))))
```

Using a for statement, we can define factorial as follows:

```
(define factorial
        (lambda (fix) ((((fix) n)))
                (for (fix)
                        (((fix) (temp 1)))
                                ((test (is (less 2) $$n) (return
$$temp))
                                (step (assign $←n (- $$n 1))))
                        (assign $←temp (* $$n $$temp))))))
```

Thus the value of (factorial 3) is 6; and the value of

(factorial (+ 2 2)) is 24

4.7 MUMBLE

Section 4.7 is logically completely separate from the rest of the language. It is not necessary to read this section to understand the rest of the document.

MUMBLE is a proposed block structured assembler for MATCHLESS which outputs relocatable binary. If a process encounters an undefined function or actor then it searches its archive file for the most recent version of the function or actor. If the search succeeds then the process calls the loader to load it and continues execution. There are two kinds of identifiers that are allowed in the language. The first kind is declared in block headers and the value is obtained by prefixing the identifier with the character '. The second kind is defined by appearing as a tag and the value is obtained by prefixing the identifier with the character :.

4.7.1 Commands

4.7.1.1 Control Structure Commands

(MUMBLE name form type argument-types declaration body) where body is of type <?> will declare a top level named block. The allowable form types are lambda and kappa.

(DPROG name declaration body) where body is of type <?> will create a block of nomenclature with name, declaration, and body.

(ENTRY name kind-of-form argument-types) will declare an entry point. The allowable kinds of forms are LAMBDA, KAPPA, and MACRO.

(GO tag) will transfer control to tag.

(; comment) where comment is of type <?> is a comment.

```
(DFOR name
        declaration
        ((INITIAL initial)
        (TEST test test-action)
        (STEP step))
        body)
```

where initial, test-action BODY, and step are of type <?> is the for statement of the language. It expands to the following where TAG is a generated symbol

```
(DPROG declaration
        initial
TAG
        (DCOND (test test-action))
        body
        step
        (GO TAG))
```

(DCOND clauses) is the conditional statement. Each clause is of the form (predicate commands).

(CALL name return-type argument-types) will create a function call to the named function.

4.7.1.2 Data Structure Commands

(' x) is the word with the indirect bit on and x in the right half.

(I a x) is the word with a in the index field and x in the right half.

('I a x) is the word with the indirect bit on, a in the index field, and x in the right half.

(= literals) is the word with left half 0 and right half the address where the multiword literal is stored.

(SPECIAL identifier type) is the address of the special cell of identifier with type.

(PATH path) where path is of type <?> is the location named by the path name. For example (path a b) is location b within block a.

(HALVES word1 word2) is the word whose left half is word1 truncated to 18 bits and whose right half is word2 truncated to 18 bits.

(SWAP word) is the word with left and right halves swapped.

(LSHIFT word1 n) is word shifted n places to the left.

(SIXBIT x) is a reference to the sixbit characters x.

(ASCIZ x) is a reference to the ascii characters x.

(BLOCK n) will allocate n words of memory.

(' theta) is the value of the MUMBLE identifier theta.

(# theta) is the value of the MUMBLE tag theta.

4.7.2 Predicates

4.7.2.1 Primitive Predicates

The primitive predicates are the conditions that can be recognized by a PDP-10 in one instruction. Predicates are used for flow of control. There are a great number of primitive predicates. We will only mention a few.

(E a x) is true if the accumulator a is Equal to the content of the effective address x

(N a x) is true if the accumulator a is Not equal to the content of the effective address x

(IE a x) is true if the accumulator a is Immediate Equal to the effective address x

(IL a x) is true if the accumulator a is Immediate Less to the effective address x

(SAE a x) Set the Accumulator a to the content of the effective address x and Equal zero

(SAN a x) Set the Accumulator a to the content of the effective address x and Not zero

4.7.2.2 Compound Predicates

(UCOND clauses) the conditional predicate.

(UAND predicates) is true if all of its predicates in turn evaluate to true.

(UOR predicates) is true if one of its predicates in turn evaluates to true.

(UNOT predicate) is true if its predicate is not true.

(SEQ commands predicate) is true when the predicate predicate is true after executing the commands.

4.7.3 Macros in MUMBLE

We can define the macro cycle which is defined to cycle the contents of three accumulators as follows:

```
(define cycle
        (macro (a b c) ((exch $$a $$b) (exch $$b $$c))))
```

Thus

(uprog () <cycle a1 a2 a3>) will expand to

```
(uprog ()
        (exch a1 a2)
        (exch a2 a3))
```

4.7.4 Examples in MUMBLE

```
(mumble factorial lambda (fix) ((fix))
        ()
        (; the argument is passed on the tuple pdl tp An "aobjn"
pointer to the tuple of arguments is passed in register ap)
        (push up u) (; save the old u)
        (move u up) (; move the pointer to the unmarked pdl into
u)
        (uprog jact
                ((fix
                        (n (i ap 1))
                        (temp (i u 1))))
                (; define n to be indexed of ap by 1 and temp to
be index u by 1.  We would use fixed point output to print out
the value of n and temp in the MUMBLE debugger.)
                (push up (= 1))
                (; push the literal 1 onto the top of the
unmarked pdl thus establishing the initial value of temp)
        again
                (ucond
                        ((seq
                                (move a 'n) (; move n into
register a and then test if the contents of a are less than 2)
                                (il a 2))
                        (move a 'temp) (; the value is returned
in a)
                        (move up u)
                        (pop u up) (; restore the unmarked pdl)
                        (popj up) (; exit the current function)
                (move a 'n)
                (imul a 'temp)
                (movem a 'temp)
                (sos 0 'n)
                (go *again)))


(mumble factorial lambda (fix) ((fix))
        ()
        (uprog factorial
                ((fix (n (i ap 1))))
                (push up u)
                (move u up)
                (ufor factorial1
                        ((fix (temp (i u 1))))
                                ((initial (push up (= 1)))
                                (test
                                        (seq
                                            (move a 'n)
                                            (il a 2))
                                        (move a 'temp)
```

```
                                        (move up u)
                                        (pop u up)
                                        (popj up) (; exit the
current function))
                        (step (sos u 'n)))
                        (move a 'n)
                        (imul a 'temp)
                        (movem a 'temp)))
```

5. PLANNER

Consider a statement that will match 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 will be 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 will be asserted. We obtain two more PLANNER statements analogous to the above by considering the contrapositive of (IMPLIES x y) which is (IMPLIES (NOT y) (NOT x)).

5.1 PLANNER Forms

5.1.1 Hierarchical Control Structure

PLANNER uses a control structure in which the hierarchy of calls is preserved so that a computation can back up to an activation through which it has already passed. The primitive functions "fail" and "failure?" enable the back track process to be controlled. The form (FAIL) will generate a simple failure which will back up to the most recently executed form (FAILURE? expression (pattern body)...) such that the pattern matches the message of the failure. For example

```
(prog (fix) (((fix) (x 3))
        (+
                (failure? $$x (() (assign $$x 4)))
                (cond
                        ((is 3 $$x) (fail))
                        (t 5))))
```

evaluates to (+ 4 5) which is 9

The identifier x is declared to be a fixed point integer which is initialized to 3. When the second argument of the all to "+" is evaluted the conditional detects that x is bound to 3 and so generates a simple failure. The failure backs up to the call to "failure?" with the message "()". 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 reset, the value is printed, and the process repeats.

5.1.2 PLANNER Functional Forms

The functional forms in PLANNER are thlambda (which is the analogue of lambda) and thkappa (which is the analogue of kappa). The syntax remains exactly the same. The sole change in the semantics is that the functional forms of PLANNER can handle the mechanism of failure.

The following example illustrates the syntax of functional forms. The function "among" which is defined below is a generally usefull PLANNER function. 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. The next example after this will give a problem that is more difficult to solve in LISP than in PLANNER. One way to assign to the identifier x the value which is the first element of the list $$1 that is greater than 5 would be (is (<?> (et $=x (greater $$x 5)) <?>) $$1). Another way would be (is $←x (larger 5 (among $$1))) where

```
(define among (thlambda ((<?> l)) (thprog (first)
again
        (thcond
                ((is ($$l) ())
                (; if l is empty generate a simple failure)
                (fail)))
        (assign ($-first $-l) ($$l)) (; set first to be the
first element of l and l to be the rest of l)
        (failure? (return $$first)
                (()
                        (; if the return fails with the message
"()" then go to again)
                        (go again))))))

(define larger (thlambda (b a)
        (thcond
                ((greaterp $$a $$b)
                        (; if a is greater than b then return a)
                        $$a)
                (t
                        (; otherwise generate a failure with the
message "()")
                        (fail ())))))
```

Thus the value of (greater 5 (among (2 4 6))) is 6.

The following is an example of a problem that is more difficult to solve in LISP than in PLANNER. The example is slightly artificial because we have not yet introduced enough of the PLANNER primitives to give a more natural example. The problem is to find the first repeated atom in an s-expression. For example "g" is the first repeated atom in ((w y z) ((a g) u q) (g q)). The MATCHLESS pattern (<contains (et (atomic) $-x)> <contains $$x>) will set the variable x to the first repeated atom. We will define (first-repeating-atom l) to be the first repeating atom of l if one exists.

```
(define first-repeating-atom
        (thlambda (l) (thprog ((<?) special) x)
```

```
                (thcond
                        ((find-x) $$l) (; if we find an x then
return it)
                        (return $$x))
                        (t (; otherwise generate a failure with
the message "()")
                        (fail ()))))))

(define find-x (thlambda (l) (thprog (answer ((?) special) x)
        (thcond
                ((is (atomic) $$l) (; if l is atomic then assign
x the value l)
                (assign $←x $$l)
                (return ())))
        (failing?
                (() (; if we are failing with the message "()",
then try again on the rest of l)
                (return (find-x (rest $$l)))))
        (assign $←answer (find-x (first $$l))) (; find an x in
the first of l)
        (thcond ($$answer (return $$answer)))
        (return (within $$x $$l)))))

(define within (thlambda (y l)
        (; the value of "within" is true only if y is an atom
within l)
        (thcond
                ((is (atomic) $$l) (; if l is atomic then it
must be y)
                (is $$y $$l))
                ((within $$y (first $$l))) (; if y is wthin the
first of y then true)
                t)
                (t (; otherwise y must be within the rest of l)
                (within $$y (rest $$l))))))
```

5.1.3 PLANNER Theorems

The following three kinds of theorems are the ones which are presently defined in the language for satisfying requests made in the body of procedures:

5.1.3.1 Consequent

(CONSEQUENT type declaration consequent body) declares that consequent is the consequent of a theorem which can be used to try to establish goals that match the pattern consequent. Whether or not the theorem will actually succeed in establishing the goal depends on the body. Typically the first action that a theorem of type consequent will take 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 function (ACHIEVE pattern properties recommendation) which is explained below. The following theorem says that if it is our goal to prove x and we know that w implies x then we should make it our goal to prove w.

```
(consequent (x w)  $?x
        (proved? (implies $?w $?x))
        (goal $$w))
```

5.1.3.2 Antecedent

(ANTECEDENT type declaration antecedent body) declares 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 antecedent has been asserted. The only way that a theorem that begins with the atom antecedent can be called is by the function (DRAW statement properties recommendation) which is explained below. 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 pattern body) can be used to try to deduce consequences from the fact that a statement that matches pattern has been erased. The only way that a function of type erasing can be called is by the function (CHANGE statement properties recommendation) which is defined below. 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 Primitive Functions

5.2.1 Data Structure Primitives

Some of the functions in PLANNER are listed below together with brief explanations of their function. Examples of their use will 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 will try to remember everything that it is doing on all levels unless commanded to forget some part of this information. In the implementation of the language special measures must be taken to ensure that identifiers receive their correct bindings. The most efficient way to implement the language is to put pointers on the stack back to the place where the correct bindings are. Value cells do not provide an efficient means of implementing the language. The default response of the language when a simple failure occurs is to back track to the last decision that it made and to make another choice.

5.2.1.1 Assertions

(DRAW statement properties recommendation) will cause PLANNER to try to draw conclusions from the statement with the properties using the recommendation to try to find an antecedent theorem (antecedent type declaration antecedent body). The value of the function draw is the value of the antecedent theorem that draws conclusions from statement. A recommendation has the form (TRY theorems) or (USE theorems). The recommendation (try th3 th1 th5 (?)) means that th3, th1, and th5 are to be tried in turn and then the theorems whose antecedents which most closely match statement are to be tried. The recommendaton (use th3 th1 (and (not (th7 th5)) (has (difficulty 9))) means that unless conclusions can be drawn using th3, th1, or some theorem except for th7 or th5 which has the difficulty 9 on its propery list, then the function draw will generate a simple failure. The recommendation (try $←x) will try any theorem which can possibly match the statement and will bind the identifier x to the name of the theorem which is used.

(draw (subset a b) () theorem5) will try to draw conclusions from the fact that the set a is a subset of the set b using theorem5. Suppose that we are keeping a global count of the number of assertions of the form (subset x y) in the global identifier count.

```
(define theorem5 (antecedent
                  (x y (((fix) common free)
count))
                  (subset $←x $←y)
                  (; x and y are locals, count is a global
fixed point number that occurs free in theorem5)
                  (; the antecedent of theorem5 is
"(subset $←x $←y)")
                  (; the following statement will add one
to count)
                  (assign $:count (+ $$count 1)))
```

(ASSERT statement properties recommendation) If the statement has already been asserted then the function assert acts as the null instruction. Otherwise, the function assert causes the statement statement with properties to be added to the data base. Then (DRAW statement properties recommendation) is evaluated. If the recommendation of the draw statement fails or if a lower level failure backs up to the assertion then statement is removed from the data base. If the null recommendation is made then the value of the function assert is the header of the assertion stored in the data base. Otherwise the value of the function assert is the value of the draw statement that it executes.

```
(assert
        (subset a b)
        ((difficulty trivial)))
```

will assert that the set a is a subset of the set b and put the value trivial under the indicator difficulty. Expressions of the form (V declaration alternatives) where alternatives is of type <?> will denote an assertion with variables declared and logical alternatives. We shall use ":" as a prefix operator to

denote variables in the quantificational calculus. We would like to emphasize that the syntax for variables in the quantificational calculus is not related to the syntax of PLANNER. For example

```
(assert
        (v (((set) :x :y :z))
                (not (subset :x :y))
                (not (subset :y :z))
                (subset :x :z)))
```

will assert in declarative form that the subset relation is transitive for sets. The function "v" is logical disjunction for clauses.

(ASSERT! statement properties recommendation) is like the function assert except that if statement has already been asserted then it will generate a simple failure instead of acting as the null operation.

(PERMANENT statement properties recommendation) is like the function assert except that statement is left in the data base even if a failure backs up to the call to the function permanent.

(TEMPORARY statement properties recommendation) is like the function assert except that statement will be withdrawn if everything succeeds in the end. In other words statement is a temporary result that will go away after we solve our current

over-all problem which is the top most call to the evaluator.

5.2.1.2 Erasures

(CHANGE statement properties recommendation) is used to try to deduce conclusions from the fact that statement no longer holds using a theorem of type erasing (ERASING type declaration pattern body). The function change is exactly analogous to the function draw.

(ERASE statement properties recommendaton) will try to find a statement in the data base that matches statement with properties. If such a statement is found then it is erased and (CHANGE statement properties recommendation) is evaluated. Otherwise the function erase acts as the null statement. If the change statement fails or if a failure backs up to the function erase, then the statement that was originally erased is restored and the whole process repeats with another statement from the data base. If the null recommendation is made then the value of the function erase is the header of the statement erased. Otherwise, the value is the value of the theorem that is used to draw conclusions from the fact that the statement was erased. The function erase is a partial left inverse of the function assert.

(erase (on-top-of brick1 brick2)) will erase the fact that brick1 is on top of brick2.

(ERASE! statement properties recommendation) is like the function erase except that if statement has not been proved then it will generate a simple failure instead of acting as the null operation.

(PERMERASE statement properties recommendation) is like the function erase except that if a failure backs up to the function permerase then it will not put the statement back in the data base.

5.2.1.3 Goals

(PROVED? pattern old-properties new-properties) tests to see if a statement with old-properties is in the data base. If there is such a statement, then the identifiers in the pattern are bound to the appropriate values and new-properties are installed as new properties of statement in the data base. If there is no such statement, then a simple failure is generated. If a simple failure backs up to the function proved?, then the identifiers that were bound are unbound and the property list is restored to its previous state. Then the whole process repeats with another statment 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 list coordinate is defined by some

atom being in some position. 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. When the bucket under some coordinate exceeds a threshold then the bucket is 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. 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. The value of the funcion proved? is the header of the assertion that matches statement.

```
(proved?
        (subset a b)
        ((difficulty trivial)))
```

will succeed only if it has been proved that a is a subset of b with the value trivial under the indicator difficulty.

(INSTANCE? pattern old-properties new-properties) is like the function proved? except that the function instance looks for a statement that can be instantiated to match the pattern. We will use ":" as a prefix operator to denote variables in the quantificational calculus. The syntax that we use for variables in the quantificational calculus is unrelated to the syntax that we use for the variables of PLANNER.

given:

```
(assert
        (v (((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:

```
(thprog (((set) w u) )
        (instance? (clause (subset $←w $←u))))
```

evaluates to ((clause (((object) :x)) (subset (1 :x) (1 :x))) difficulty hard)

w gets the value (f :x)

u gets the value (1 :x)

Suppose that we know that a is a subset of b or a is a subset of c. In other words we assert (v () (subset a b) (subset b c)). evaluate:

```
(thprog (((set) x))
        (instance? (clause (subset a $←x))))
```

x gets the value (either b c)

In other words x is either b or c

(ACHIEVE goal properties recommendation) will attempt to achieve goal using a consequent theorem (CONSEQUENT

declaration consequent body) with according to recommendation. The goal must match the consequent.

(GOAL pattern properties recommendation) the first thing that the function goal does is to evaluate (PROVED? pattern properties). If the evaluation produces a failure then the value of the function goal is (ACHIEVE pattern properties recommendation).

given: (subset a b)

evaluate: (thprog (((set) x y))

(goal (subset $←x $←y)))

x gets the value a

y gets the value b

(GOALS) returns as its value a list of the currently active goals.

5.2.2 Control Structure Primitives

(THVAL expression bindings state) will evaluate expression with bindings and local state. At any given time PLANNER expressions are being evaluated in a state. A top level a process begins by using the global data base as its state. It can switch into a local state by using the function stateprog or the function thval. This local state determines what changes have been made to the data base i.e. what erasure, assertions, definitions of theorems have been made since the last time that

the data base was updated. States are stored as a linear list of changes to the data base. Thus there can be several incompatible states of the world simultaneously under consideration. However, the use of local states slows up data base manipulations since elements of the local state must be searched linearly.

(STATE) returns as its value the current local state.

(UPDATE state) will update the data base according to state.

(THCOND type clauses) where clauses is of type segment evaluates the first element of each clause in turn to try to find one that doesn't cause a failure or return () as a value. If such a clause is found then the remaining elements of the clause are evaluated in turn.

(thcond (t (fail))) will fail with the message ()

(thcond ((fail) 3) (t 7)) evaluates to 7

(thcond (() 3)) will evaluate to ()

(thcond (() 3) (t 4)) evaluates to 4

(thcond (t (fail)) (t 5)) fails

(ATTEMPT type clauses) where clauses is of type segment will attempt to find one whole clause which can be successfully evaluated. The function attempt is very much like the the function thcond. The main difference is the function thcond will not try the remaining cluases if a failure occurs in a

clause after the predicate 1or the clause has been evaluated.

(attempt (t (fail))) will 1ail with the message ()

(attempt ((1ail) 3) (t 7)) evaluates to 7

(attempt (() 3)) will evaluate to ()

(attempt (() 3) (t 4)) evaluates to 4

(attempt (t (fail)) (t 5)) evaluates to 5

(THPROG type declaration progbody) where progbody is of type segment is like the MATCHLESS function prog except that it can handle the mechanism of failure.

(RETURN expr) causes the value o1 expr to be returned.

(TEMPROG type declaration progbody) is like the function thprog except all assertions and erasures that are made within the scope of the function temprog must be undone when the function temprog is exited. The 1unction temprog is use1ul 1or dealing with hypotheticals. If we know that a 1ormula o1 the form (CLAUSE x y) is true and we want to establish a goal o1 the form g then we could write:

```
(THPROG ()
        (TEMPROG ()
                 (ASSERT x)
                 (GOAL g))
        (TEMPROG ()
                 (ASSERT y)
                 (GOAL g))
        (ASSERT g))
```

The above form of disjunction elimination is often used when y is o1 the form (NOT x). Goals of the 1orm (CLAUSE x y) can be established as follows:

```
(THPROG ()
        (TEMPROG ()
                 (ASSERT (NOT x))
                 (GOAL y))
        (ASSERT (V () x y))
```

(STATEPROG type declaration body) where body is of type segment is like the function thprog except that within the function stateprog assertions, erasures, and the definitions of theorems are made in the current local state instead of in the global data base.

(THAND conjuncts) where conjuncts is of type segment is like the LISP function and except that the function thand can handle the mechanism of failure. (thand conjunct/1 ... conjunct/n) is equivalent to

```
(thcond
        (conjunct/1
                .
                .
                .
                (thcond
                        (conjunct/n)
                        (t (fail))))
        (t (fail)))
```

(THOR disjuncts) where disjuncts is of type segment is like the LISP function or. (thor disjunct/1 ... disjunct/2) is equivalent to (thcond (disjunct/1) ... (disjunct/2) (t (fail)))

(THNOT x) is an abbreviation for (thcond (x (fail)) (t t)). Thus (thnot ()) is t, (thnot t) is (), and (thnot (fail)) is t. The function thnot is due to T. Winograd.

5.2.2.1 Failure Primitives

(UNIQUE) will fail if the current goal is not unique among all the goals that are currently active.

(FAIL) causes a simple failure to be reported above. PLANNER will reconsider the last decision that it made. If there are any alternatives, it will chose one and continue execution.

(FAIL point) causes a failure to point. For example if point is "theorem" then the function fail will cause the current theorem to fail. If point is "goal" then it will cause the current goal to fail.

(FAIL point message) acts exactly like (fail point) except that once it has failed back to the point then it converts to a failure with a message which can be caught only by the functions failure? or failing? which are explained below.

(FAILURE? expr fail-clauses) where fail-clauses is of type segment evaluates expr. If the evaluation does not produce a failure then the value of the function "failure?" is the value of expr. If the message of the failure matches the first element of a clause then the rest of the elements of the clause are evaluated. Otherwise the failure continues to propagate upward.

(failure? (fail)

(() hello)) will evaluate to hello

(FAILING? fail-clauses) where fail-clauses is of type segment will act as the null operation unless a failure backs up to it. If a failure backs up to it then it acts like the function "failure?".

```
(thprog ()
        (failing? (() (return a)))
        (fail)) evaluates to a
```

(FAIL-TO tag) causes failure to a tag which must previously have been passed over. Execution resumes with the statement after the tag.

```
(thprog (a)
        (assign $←a 3)
there
        (thcond
                ((is 4 $$a)
                (return $$a)))
        (assign $←a 4)
        (fail-to there)) evaluates to 4
```

(FAIL-PAST tag) causes a failure to tag which previously must have been passed over and then the generation of a simple failure.

```
(thprog (a)
        (failing? (() (return $$a))
where
        (assign $←a 8)
        (fail-past where)) evaluates to 8
```

(SUCCEEDING? declaration body) where body is of type segment will act as the null statement unless the remaining compuation succeeds. In case of success the declaration is activated and the body is executed.

5.2.2.2 Finalize primitives

(FINALIZE-TO tag) causes all actions that have been taken since tag was passed over to be finalized so that if the compuation later fails they will not be undone. Finalization is mainly used to save storage. The next statement to be executed is the one immediately after the call to the function finalize-to.

(FINALIZE point) causes all actions that have been taken since point was passed to be finalized. For example (FINALIZE goal) will finalize all actions that have been taken since the last goal.

5.2.3 Repitition Primitives

```
(THFOR type declaration
        ((INITIAL initial)
        (PROVED pattern old-properties new-properties)
        (TEST test test-action)
        (FINAL final-action)
        (STEP step)
        (LIST element condition))
        body)
```

where body is of type segment is the for statement of PLANNER. For each assertion in the data base that matches pattern with old-properties, the statement is given new-properties and an attempt is made to execute the body. For example the following statement will place all the bricks on brick1 in the blue box.

```
                (thior

                                        (((brick) x))

                                        ((proved (on-top-of $←x
brick1)))

                                (pick-up $$x)

                                (place-in (' (blue box))))

        (PERSIST type declaration
                ((INITIAL initial)
                (TEST test test-action)
                (LIST item condition)
                (STEP step)
                (FINAL final))
                body)
```

where body is of type segment is equivalent to the following:

```
                (THPROG type (declaration ((xpr) (COLLECTED ()))
                        (; declare COLLECTED to be an s-
expression and initialize it to nil)
                        (FAILING? (() final (RETURN
$$COLLECTED)))
                        body
                        (THCOND (test test-action (RETURN
$$COLLECTED)))
                        (THCOND
                                (condition
                                (; if the condition is met then
add item to the end of COLLECTED)
                                (ASSIGN $$COLLECTED (<$$
COLLECTED> item))))
                        step
                        (FAIL) (; generate a simple failure))
```

"Are all the blocks in box1 green?" will translate to

```
                (persist (((block) b))
                        ((final (return t)))
                        (goal (in $←b box1)) (; find a block in
box1)
                        (thcond
                                ((goal (green $$b)) (; if the
block is green then continue with the loop))
                                (t
```

```
                                    (fail persist) (;otherwise
generate a failure out of the persist loop))))

        (FIND
                (BETWEEN lower upper SUCCEED)
                declaration
                item
                body)
```

will find between lower and upper items according to the body. The function find does not consider possible interactions between the elements sought. The find primitive function is equivalent to the following:

```
                (PERSIST (declaration ((FIX) (NUMBER 0)))
                        ((TEST (ASSIGN? $$NUMBER upper)) (; if
we have found at least upper items then we are done)
                        (LIST item) (; we will make a list of
the items that we find)
                        (STEP (ASSIGN $$NUMBER (+ $$NUMBER 1)))
(; after each pass through the loop, we will add one to NUMBER)
                        (FINAL (THCOND  ((ASSIGN? (LESS lower)
$$NUMBER) (FAIL)))) (; as our final action we will test to see
that we have collected at least lower items If not then generate
a simple failure))
                        body)

                "Find three boxes that contain green blocks."
```

will translate to:

```
                (find 3 (((box) x) ((block) b)) $$x
                        (goal (box $=x))
                        (goal (contains $$x $=b))
                        (goal (green $$b)))
```

5.2.4 Co-routine Primitives

In more complicated situations, we find that it is convenient to be able to have more than one PLANNER process.

(THCREATE x) will create a PLANNER process which will begin evaluation with the function call x. The value of the function thcreate is the name of the created PLANNER process.

(THRESUME process expression) will resume execution of process from the point that control last left it. The value of expression is made the value of the call to "thresume" that last caused control to leave process.

(THPASSON process expression) will resume (see above) the process and then cause the calling process to die.

(COFAIL process message) will generate a failure with a message within process at the last point that execution left the process.

```
(EXHAUST type declaration
        ((INITIAL initial)
        (TEST test test-action)
        (ACTION action)
        (LIST item condition)
        (STEP step)
        (FINAL final))
        body)
```

where body is of type segment will attempt to execute body once for each time that action is successfully evaluated. Every time that the body it executed the function exhaust will send a simple failure to the action to see if it has any alternatives. An "exhaust" loop The function is very much like a "persist" loop which is defined above. Both kinds loops are driven by the

failure mechanism. The main difference is that the effects of executing the body of a "persist" loop are not preserved because a failure must propagate through the body before it can be executed again. In an "exhaust" loop a separate process is created for the action so that the effects of executing the body can be preserved. The function exhaust is equivalent to the following expression:

```
(THPROG type
    (
        ((xpr) (COLLECTED ()))
        ((proc)
                (CURRENT (CURRENT))
                (ACTION-PROCESS (THCREATE (ACTION-FUNCTION
$$CURRENT))))
        (; declare COLLECTED to be a s-expression initialized to
(); CURRENT to be initialized to the name of the current
process; ACTION-PROCESS to be initialized to the name of a new
process which begins exection with the call (ACTION-FUNCTION
$$CURRENT) which will pass the name of the current process to
the created process)
        (THCOND
                ((IS EXHAUSTED (THRESUME $$ACTION-PROCESS))
                (; start the PLANNER process $$ACTION-PROCESS in
which the action will be executed; if the current process is
resumed with the value EXHAUSTED then go to the tag EXHAUSTED;
the latter will happen only if the action fails before
successfully evaluating even once)
                 (go EXHAUSTED)))
CONTINUE
        (THCOND (test test-action (RETURN $$COLLECTED))) (; if
the test is met then execute the test-action)
        body
        (THCOND
                (condition
                (ASSIGN $$COLLECTED (<$$ COLLECTED> item)))) (;
if the condition is met then add the item to the end of the list
of collected items)
        (; the expression (COFAIL $$ACTION-PROCESS) will suspend
execution of the current process and will begin failing from the
point within the action process where execution last left off)
        (THCOND
                ((IS EXHAUSTED (COFAIL $$ACTION-PROCESS))
```

```
                (; if the current process is resumed with the
value EXHAUSTED then go the location EXHAUSTED)
                (GO EXHAUSTED))
        (GO CONTINUE)
EXHAUSTED
        final
        (RETURN COLLLECTED))
```

The following function is defined so that we can start off the evaluation of the action process.

```
(define ACTION-FUNCTION
    (THLAMBDA (((proc) MAIN))
        (FAILING? ((?) (THPASSON $$MAIN EXHAUSTED)))
        (; when the action finally is exhauseted resume the
process $$MAIN with the value EXHAUSTED and kill the action
process)
        action
        (THRESUME $$MAIN SUCCESS)
        (; resume the main process with the value SUCCESS)))
```

Suppose that we have a way to generate the elements of a set w. For each element of w, we want to deduce consequences from the fact that it has property q. Then we want to try to show that w has the property q.

```
        (thprog ()
                (exhaust (((set) x))
                                ((action (goal (subset $-x w))))
                        (assert (q $$x))
                (goal (q w)))
```

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 produce a large number of extraneous clauses. 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. In order to do this we would need the following functions:

(RESOLVE (pat1 pat2) resolvent new-property-list)) will result in resolving all clauses that match the pattern pat1 with all clauses that match pat2 in all possible ways to yield a clause which must match the pattern resolvent. The resolvent will be stored in the data base with new-property-list.

(RESOLVE1 (pat1 pat2) result new-property-list)) will resolve a clause that matches the pattern pat1 with one that matches the pattern pat2 and assign the pattern result to the result. If there are no such clauses then a simple failure is generated. If a simple failure backs up to the function resolve1 then it trys again with a different pair of clauses.

```
(FOR-RESOLVENT type declaration
        ((INITIAL initial)
        (CLAUSES pat1 pat2)
        (FINAL final)
        (RESOLVENT result new-property-list)
        (LIST element condition)
        (STEP step))
        body)
```

where body is of type segment will attempt to execute the body of the for statement once for each result of resolving a clause that matches the pattern pat1 with a clause that matches the pattern pat2.

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" uniform proof procedure should be easy in PLANNER. However, we should not rely on a uniform proof procedure to

solve our problems for us.

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 (lambda (bool) (((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)
```

```
(define backward
(consequent (((set) x y z)) (subset $?x $?z)
        (unique) (; the current goal must be unique)
        (goal (subset $?x $?y) () (try backward (?)))
        (goal (subset $$y $?z) () (try backward))
        (assert (subset $$x $$z) () ?))))
```

Now if we ask PLANNER to evaluate (goal (subset a c)) then we will obtain the following protocol:

```
(goal (subset a c))
        (proved? (subset a c))
        fail
        (achieve (subset a c))
        enter backward
        x becomes a
        z becomes c
        (unique)
        (goal (subset a $?y))
                (proved? (subset a $?y))
node 1,9
        y becomes a
        (goal (subset a c))
                (proved? (subset a c))
                fail
                (achieve (subset a c))
                enter backward
                x becomes a
                z becomes c
                (unique)
                (goal (subset a $?y))
                        (proved? (subset a $?y))
                        fail
                        (achieve (subset a $?y))
                        enter backward
                        x becomes a
                        z becomes $?y
                        (unique)
                        fail
                fail
node 1,9  ;note that this node appears above
        y becomes b
        (goal (subset b c))
                (proved? (subset b c))
        (assert (subset a c))
        succeed

After the evaluation the data base contains:
        (subset a b)
        (subset a a)
        (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 a, so it binds y to a. Next (subset a 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 a, and z is bound to c. The subgoal (subset a $?y) is established causing backward to again be called recursively with x bound to a 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 a $?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 a c) in the first place. Thus y is bound to b instead of a. Now the system sets the 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. 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, epsilon

naught, 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).

5.4.2 Using an Antecedent Theorem

Suppose we give PLANNER only the following theorems.

```
given:
        (subset a b)
        (subset c a)

(define forward-right
        (antecedent (((set) x y z)) (subset $-y $-z)
                (goal (subset $?x $$y))
                (assert
                        (subset $$x $$z)
                        ()
                        (try forward-right forward-left))))

(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) () ?), we will 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))
                (proved? (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))
                (proved? (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))
                        (proved? (subset $?x a))
                        fail
                enter forward-left
                x becomes a
                y becomes d
                (goal (subset d $?z))
                        (proved? (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:

```
(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 will generate a unique symbol which has the type of its argument. On entrance to the function temprog the identifier x will be bound to a freshly created symbol. The above theorem is a constructive analogue of

```
(all (lambda (boole)(((set) :a) ((set) :c))
        (implies
                (all (lambda
                        (boole)
                        (((object) :x))
                        (implies (element :x :a)(element :x
:c)))
                (subset :a :c)))
```

Going in the opposite direction, we have

```
(define th4-5 (antecedent
        (((set) a b))
        (subset a b)
        (assert (theorem (antecedent
                (((element) x))
                (element $?x $?a)
                (assert (element $?x $?b) () ?)))))

(define th4-6 (antecedent
        (((set) a b))
        (subset a b)
        (assert (theorem (consequent
                (((element) x))
                (element $?x $?b)
                (goal (element $?x $?a)))))))

(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 (lambda
        (boole)
        (((object) :x) ((set) :s))
```

```
        (implies
                (some (lambda
                        (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 (lambda
        (boole)
        (((set) :a) ((set) :b) ((set) :c))
        (implies
                (and (subset :a :b) (subset :b :c))
                (subset :a :c))
```

Often we will 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 pattern function clause will be used to match clauses. It will use the fact that disjunction is commutative and associative. The pattern function unify will be used as a varient of the pattern function clause in which the clauses to be unifyed will be given as the first element of the function unify. We will 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))

(define necessary
  (antecedent
        (((set) a b) ((object) x)clause)
        (let (clause (subset $~a $~b) <?>) $~clause)
                (resolve
                        ($$clause
                        (unify ()
                                ((not (subset $$a $$b)))
                                (not (element $?x $$a))
                                (element $?x $$b)))))))
```

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.

```
(define sufficient
  (antecedent
        (((set) a b) clause)
        (let (clause (not (subset $~a $~b)) <?>) $~clause)
        (resolve
                ($$clause
                (unify ()
                        ((subset $$a $$b))
                        (element (element-of-difference $$a $$b)
$$a))))
        (resolve
                ($$clause
                (unify ()
                        ((subset $$a $$b))
                        (not (element
                                (element-of-difference $$a $$b)
                                $$b)))))))
```

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:

```
(thprog ()
        (temprog (((set)
                        (a (arbitrary (set)))
                        (b (arbitrary (set)))
                        (c (arbitrary (set)))))
                 (assert (v () (subset $$a $$b)  () ?)
                 (assert (v () (subset $$b $$c)  () ?)
                 (assert (v () (not (subset $$a $$c) () ?)
                 (goal (resolve ())))
        (assert (v (((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 b)) (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.
```

6. More on PLANNER

6.1 PLANNER EXAMPLES

6.1.1 London's Bridge

Most of the time we decide which statements that 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 a 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)
                (thcond
                        ((erase
```

```
                                    (at $-other-brick $$place  (add1
$$height)) ?) (; 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. 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
        (
                ((object) a c)
                ((functor (?) (?)) predicate)
                (<?> argsa1 argsa2 argsc1 argsc2))
        (analogous $?a $?c $?predicate)
                (unique) (; the current goal must be unique)
                (thcond
                        ((proved? (test-analogous $?a $?c))
                        (; if a and c are test-analogous then we
are done)
                        (return ())))
                (proved? (a-object $?a))
                (proved? (c-object $?c))
                (; find an a-object and a c-object)
                (temporary (test-analogous $$a $$c $?predicate))
(temporarily assert that a and c are test-analogous)
                (proved? ($?predicate $←argsa1 $$a $←argsa2))
                (proved? ($$predicate $←argsc1 $$c $←argsc2))
                (; find a predicate in which both a and c are
arguments)
                (thcond
                        ((is (non ()) ($$argsa1))
                        (goal (corresponding-analogous
($$argsa1) ($$argsc1) $$predicate))))
                (thcond
                        ((is (non ()) ($$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)
                (thcond
                        ((proved? (test-analogous $?a $?c
$?predicate))
                        (; if a and c have already been asserted
to be test-analogous then we only have to look at the rest of
the elements)
                        (go rest)))
                (proved? (analogous $?a $?c $?predicate))
```

```
        rest
                (thcond
                        ((is (non ()) $$aa)
                        (proved? (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)
                (proved? (test-analogous a1 $?c inside))
                FAIL
        (proved? (a-object a1))
        (proved? (c-object $?c))
node 1
        c gets the value c2
x gets the value c2
        (temporary (test-analogous a1 c2 inside))
        (proved? (inside a1 a2))
        (proved? (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))
        (proved? (inside c1 c2))
        (goal (corresponding-analogous (a2) (c2) inside))
                enter chop-off-another
                a gets the value a2
                c gets the value c2
                        (proved? (test-analogous a2 c2 inside))
                        FAIL
                (proved? (analogous a2 c2))
                        enter pair
                        a gets the value a2
                        c gets the value c2
```

```
                        (unique)
                                (proved? (test-analogous a2 c2
inside))
                                FAIL
                        (proved? (a-object a2))
                        (proved? (c-object c2))
                        (temporary (test-analogous a2 c2
inside))
                        (proved (inside a1 a2))
                        (proved (inside c1 c2))
                        (goal (correspoinding-analogous (a1)
(c1) inside))
                                enter chop-off-another
                                a gets the value a1
                                c gets the value c1
                                (proved? (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's 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
  (lambda (x)
        (cond ((is () $$x) 0)
              ((is (atomic) $$x) 1)
              (t (+
                    (number-of-atoms (1 $$x))
                    (number-of-atoms (rest $$x)))))))

(define list-of-atoms
  (lambda (x)
        (cond ((is () $$x) ())
```

```
                ((is (atomic) $$x) ($$x))
                (t (append
                        (list-of-atoms (l $$x))
                        (list-of-atoms (rest $$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. W. Bledsoe has suggested that still another example of analogous proofs is found in the Schwartz inequality:

```
 (not (greater
        (**
                (+
                        (* (x 1) (y 1))
                        (* (x 2) (y 2)))
                2)
        (*
                (+
                        (** (x 1) 2)
                        (** (x 2) 2))
                (+
                        (** (y 1) 2)
                        (** (y 2) 2))))))

(not (greater
        (**
                (sigma 1 n (lambda (real) (((fix) i)) (** (* (x
i) (y i)) 2)))
                2)
        (*
                (sigma  1 n (lambda (real) (((integer) i))(** (x
i) 2)))
                (sigma 1 n (lambda (real) (((integer) i)) (** (y
i) 2))))))

(not (greater
        (** (integral (* f g)) 2)
        (*
                (integral (** 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 (((functor (boole) ((integer)))
p))
        (all $~p)
                (temprog (((integer) (n (arbitrary (integer)))
))
                        (goal ($$p 0))
                        (assert ($$p $$n))
                        (goal ($$p (+ $$n 1))))
                (assert (all $$p))))
```

The type (functor (boole) ((integer))) is the type of a function which returns a boolean value and has one argument which is a fixed point number. If we are given the facts (= (+ 0 0) 0) and

```
(clause (((integer) x y))(= (+ :y (+ :x 1)) (+ (+ :y :x) 1))
```

then we can establish

```
(all (lambda (integer) (((integer) :n)) (= (+ 0 :n) :n))).
```

The following theorem will do induction on s-expressions:

```
(define expr-induction
(consequent
        (
                ((functor (boole)((expr)))
                p))
        (all s~p)
                (temprog
```

```
                    (((expr) (a (arbitary (atom)))
                     (car (arbitrary (expr)))
                     (cdr (arbitrary (expr)))))
          (goal ($$p $$atom))
          (assert ($$p $$car))
          (assert ($$p $$cdr))
          (goal ($$p (cons $$car $$cdr))))
(assert (all $$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

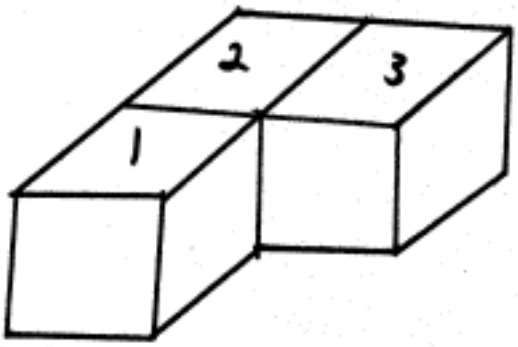
PLANNER can be used to find objects from partial or schematic descriptions. The statement (perpendicular (line $←a $←b) (line $←c $←d)) will be defined to mean that the lines (line $$a $$b) and (line $$c $$d) are perpendicular. The MATCHLESS function (HASVAL? arg) tests to see if the identifier arg has a value. The value of (genbraces) is () and the value

of (genbraces 1) is (1). 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))
    again (goal (glued $$a (et (non $$b) $?c)))
          (goal (orthogonal (line $$a $$b) (line $$a $$c)))
          (thcond ((thor
               (goal
                    (glued $$a (et (non $$b) (non $$c))))
               (goal (glued $$b (non $$a)))
               (goal (glued $$c (non $$a))))
          (tail-past again)))))
```

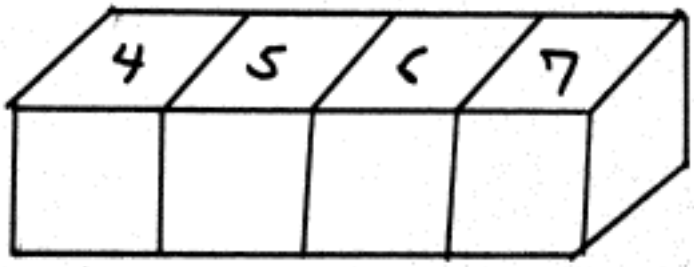
The description can be used in the obvious way to find three-corners. The statement (stick $←a $←b) is defined to mean that $$a and $$b are end bricks of a line of bricks and (between $←a $←b $←c) is defined to mean that brick $$b is between bricks $$a and $$c. Examples of sticks are shown are shown in figure 1.

```
(define find-stick
        (consequent
               (((brick) a b) ((fix) n))
               (stick $?a $?b $←n)
                      (proved? (brick $?a))
                      (proved? (brick $?b))
                      (goal (stick-segment $$a $$b (difference
```



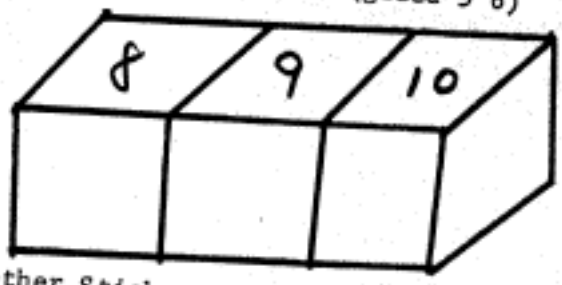
A Three-Corner:

(cube 1)
(cube 2)
(cube 3)
(glued 1 2)
(glued 2 3)



A Stick:

(cube 4)
(cube 5)
(cube 6)
(cube 7)
(glued 4 5)
(glued 5 6)
(glued 6 7)



Another Stick:

(cube 8)
(cube 9)
(cube 10)
(glued 8 9)
(glued 9 10)

```
$$n 2)))
                        (assert (stick $$a $$b $$n))))

(define find-stick-segment
 (consequent
 (((brick) x y w)((fix) n))
 (stick-segment $?x $?y $?n)
        (thcond
                ((minusp $$)
                (thfail))
                ((goal
                        (glued $?w $?x)
                (goal (orthogonal (line $$x $$w) (line $$x
$?y)))
                (thfail)))
                ((goal (glued $?x $?y))
                (thcond
                        ((thand
                                (goal (glued $?w $?y))
                                (goal (orthogonal (line $$y $$w)
(line $$y $$x)))
                        (thfail)))
                (return ())))
        (goal (glued $?w $$x))
        (goal (between $$x $$w $$y))
        (goal
                (stick-segment $$w $$y (sub1 $$n))
                ()
                (try find-stick-segment (?))))))
```

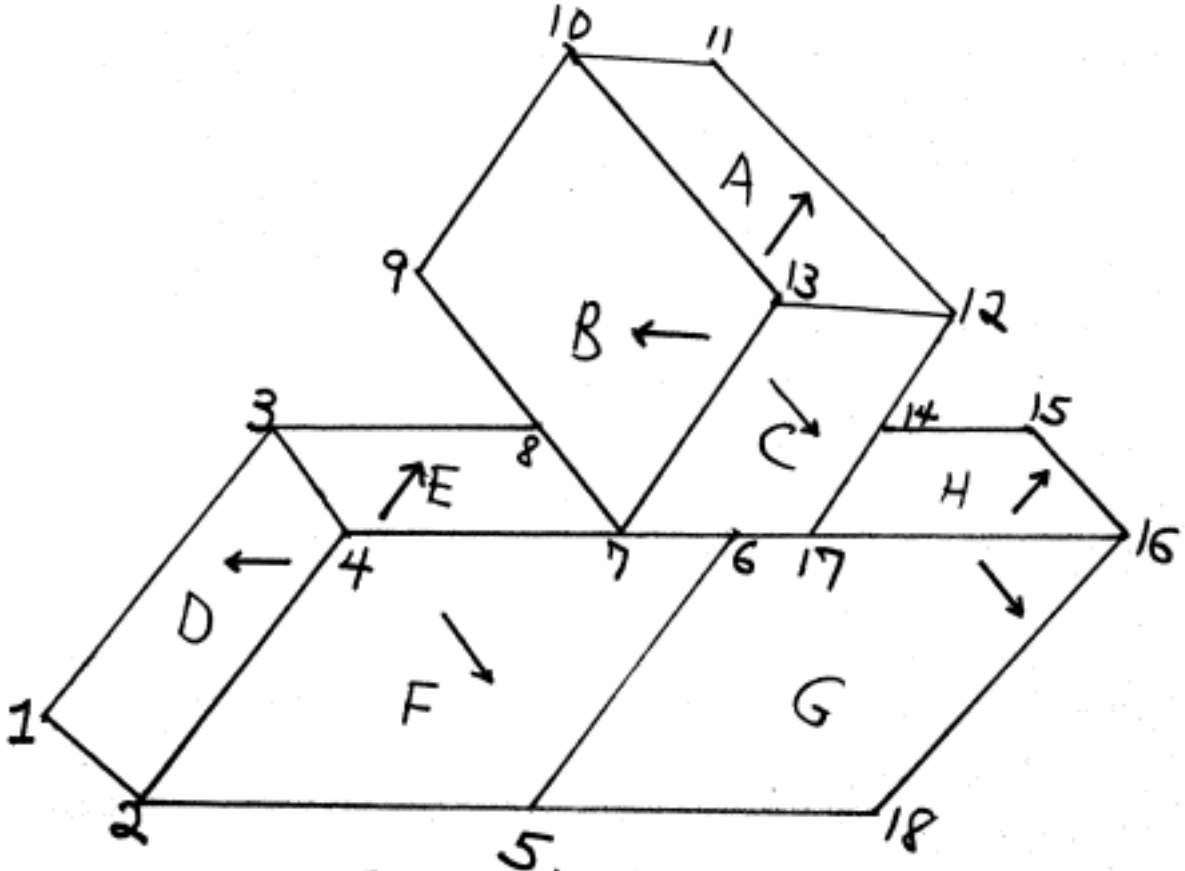
6.1.4.2 Constructing Examples of Descriptions

Given a description of a structure (such as a stick) we would like to bable 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.

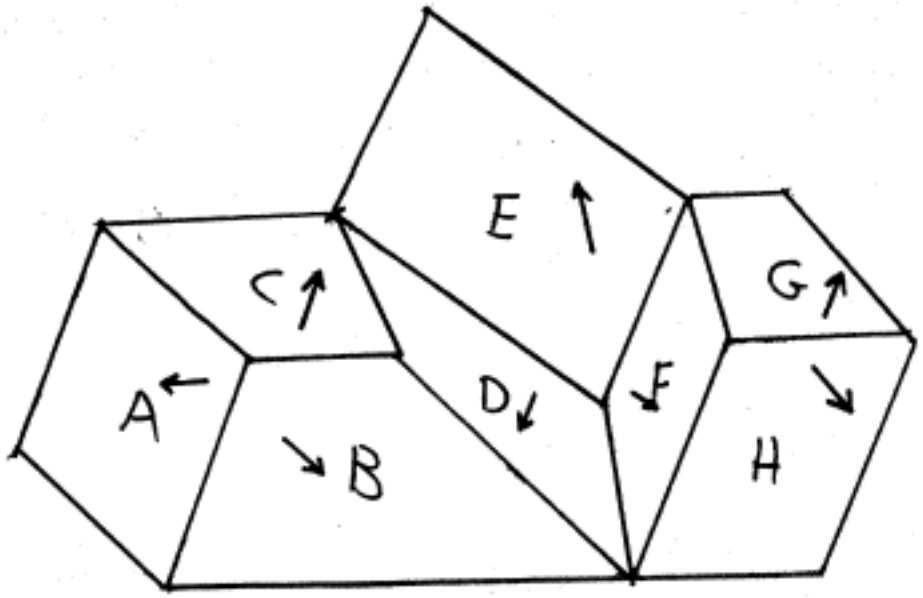
```
(define make-stick (consequent
        (((brick) x y w) ((fix) n))
        (make-stick $←x $←y $←n)
                (thcond ((lessp n 3)
                        (glue (bottom $$x) (top $$y))
                        (return ())))
                (is $←w (new-brick))
                (glue (bottom $$x) (top $$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. Theorem proving techniques have the advantage that they can take into account very general kinds of information. 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 constructing 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



Example 1



Example 2

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.

(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))
(adjacent a b)
(adjacent a c)
(adjacent b c)
(normal c (direction 10 13))
(normal d (direction 7 4))
(normal e (direction 2 4))
(adjacent d e)
(normal f (direction 3 4))
(adjacent d f)
(adjacent e f)
(normal h (direction 16 18))
(normal g (direction 15 16))
(adjacent g h)
```

The following statement about example 1 satisfiable:

```
(and (resting-on (brick a b c) (brick e 1 d)) (resting-on
(brick a b c) (brick g h)))
```

The following statements about example example are valid:

```
(adjacent a c)
(adjacent a b)
```

```
(adjacent b c)
(normal a (direction 12 14))
(normal c (direction 3 14))
(adjacent g h)
(normal g (direction 5 6))
(normal h (direction 8 6))
(not (adjacent  c a))
(not (adjacent b a))
(adjacent d e)
(adjacent e f)
(adjacent d f)
(normal e (direction 4 13))
(normal d (direction 9 13))
(normal f (direction 11 13))
```

The following statement about figure2 s 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 co-ordinates 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 provable. 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 Guzman's program 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.

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 deepening 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 could 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 (lambda (x)
        (cond
                ((is (atomic) $$x)
                $$x)
```

```
(t
(!(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. A more powerful wait primitive would make synchronization easier. If a process calls (wait predicate) then its execution will be blocked until the predicate becomes true.

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 resolved:

```
        (is-smaller-than hand (' (pig pen)))

(define example-of-bar-hillel
 (antecedent
         (((object) x y))
        (in $-x $-y)
                (thcond
                        ((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 deductive logic 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 ackward becasue natural language lacks many oi the descriptive and procedural primitives of PLANNER.

Current Problems and Future Work

We shall call the logistic system based purely on the primitives of PLANNER "robot logic". Robot logic is a kind of hybrid between the classical logics such as the quantificational calculus and intuitionism, and the recursive functions as represented by the lambda calculus and Post productions. The model theoretic definition of truth in robot logic is complicated by the existence of the primitive erase and the whole PLANNER interpreter. The semantics of PLANNER theorems are most naturally defined dynamically by the properties of procedures. The semantics of theorems in the quantificational calculus can be defined by models of possible worlds. In comparison with the quanditificational 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 doesn't 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.

Memory: There is never enough fast random access storage.

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 cons up 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:

```
(thfor (((face) face1 face2) ((brick) brick))
               ((proved (small-brick $←brick)))
        (proved? (face $←face1 $$brick))
        (proved? (color $$face1 green))
        (proved? (face $←face2 $$brick))
        (proved? (color $$face2 yellow)
        (pick-up $$brick)
        (carry-to (above (' (red box))))
        (drop))
```

T. Winograd has developed a program to translate English into PLANNER theorems. An interestiong expreriment 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. M. 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.

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 deepening: 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 doesn't 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 because 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 contradicton in PLANNER 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. Another type of problem that PLANNER will not solve very naturally is given a predicate p defined in the first order predicate calculus to show nonconstructively that there is some object x such that (p x) is true.

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 itself, it needs:

1: a way to express properties of its procedures.

2: A way to establish that the properties do in fact hold for its procedures.

We shall express the properties of an expression x by the following function.

(INTENT 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 varialbles in upper case. Also we shall use !(and)! as meta-braces for (and) respectively. For example the intentions in the prog below are all true.

```
(prog (x) (((x) (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))
                (assign $$b (+ $$b 1))
                (thlambda (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)
        (return $$b))
```

The essential idea for intentions comes from the break function of LISP introduced by W. Martin. The expression (INTENTION pattern exp) will be used to express the fact that the pattern must match the value of exp. 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 declaration predecessor expression function successors) is exactly like the function intent except that intention variables can be declared in the declaration. In addition we need a function (OVERALL declaration predecessor expression function successors) which is

exactly like the function "intend" except that it is used to state the overall intention of a procedure. All the intentions in the function fact are true where

```
(define fact (lambda (fix) ((fix) n))
(overall ()
     (goal (not (lessp $$n 0)))
     (intent
        (assert (not (lessp $$n 0)))
        (prog (fix) ((fix) (temp 1) (i $$n)))
                (intent (goal (= $$temp 1)))
                (intent (goal (= $$i 1)))
        again
                (overall ()
                        (goal (= $$temp !(factorial $$i)!))
                (intent
                        (assert (= $$temp (factorial $$i)))
                (cond
                        ((is 0 $$i)
                        (intent (thcond
                                ((goal (= $$n 0))
                                (goal (= $$temp 1)))
                                ((goal (not (= $$n 0)))
                                (goal (= $$temp !(* $$n
!(factorial !(- $$n 1)!)!)!))))))
                        (intent (goal (= $$temp !(factorial
$$n)!)))
                        (return $$temp)))
                (intent (goal (= $$temp !(factorial $$i)!)))
                (assign $:temp (* $$temp $$i))
                (assign $:i (- $$i 1))
                (go again))
        (thlambda (X)
                (goal (= $$X !(factorial $$n)!)))
     (thlambda (X) (assert (= $$X !(factorial $$n)!))))
```

where

```
(define factorial (lambda (fix) ((fix) n))
(overall ()
     (goal (not (lessp $$n 0)))
     (intent
        (assert (not (lessp $$n 0)))
        (cond
                ((is 0 $$n) 1)
                (t (* (factorial (- $$n 1)) $$n)))
        (thlambda (X)
```

```
                (temprog ()
                        (assert (= $$n 0))
                        (goal (= $$X 1)))
                (temprog ()
                        (assert (not (= $$n 0)))
                        (goal (= $$X !(* $$n !(fact !(- $$n
1)!(!)!)))
                (goal (= $$X !(fact $$n)!)))
     (thlambda (X)
        (assert (theorem (antecedent ()
                (= $$n 0)
                (assert (= $$X 1)))))
        (assert (theorem (antecedent ()
                (not (= $$n 0))
                (assert (= $$X !(* $$n !(fact !(- $$n
1)!)!)!)))))
        (assert (= $$X !(fact $$n)!))))
```

The intentions for the function fact' defined below are not so easy to establish.

```
(define fact' (lambda (fix) (((fix) n))
(overall (((fix) (ARG $$n))
     (goal (not (lessp $$n 0)))
     (intend (((fix) (ARG $$n))
        (assert (not (lessp $$n 0)))
        (prog (fix) (((fix) (temp 1)))
                (intent (goal (= $$temp 1)))
                (; test to see if the identifier temp was really
intialized to 1)
          again
                (overall ()
                        (goal (= $$temp !(combinations $$arg
$$n)!)))
                (intent
                        (assert (= $$temp !(combinations $$arg
$$n)!)))
                (cond
                        ((is 0 $$n)
                        (intent  (= $$temp !(factorial $$ARG)!))
                        (return $$temp)))
                (intent (greaterp $$n 0))
                (assign $:temp (* $$temp $$n))
                (assign $:n (- $$n 1))
                (go again))
```

```
        (thlambda (X) (goal (= $$X !(factorial $$ARG)!)))
     (thlambda (X) (assert (= $$X !(factorial $$ARG)!))))))

(define combinations (fix) (((fix) n)((fix) r))
(overall ()
     (thand
        (goal (not (lessp $$n 0)))
        (goal (not (lessp $$r 0)))
        (goal (lessp $$r $$n)))
     (intent
        (thand
                (assert (not (lessp $$n 0)))
                (assert (not (lessp $$r 0)))
                (assert (lessp $$r $$n)))
        (cond
                ((is $$n $$r) 1)
                (t (* $$n (combinations (- $$n 1) $$r)))
        (thlambda (X)
                (thcond
                        ((goal (= $$r 0))
                        (goal (= $$X !(factorial $$n)!))))))
     (thlambda (X)
        (thcond
                ((goal (= $$r 0))
                (assert (= $$X !(factorial $$n)!)))))))
```

We can define the data types of LISP and write intentions for the LISP primitives. The type "xpr" is the type s-expression.

```
(define xpr (type () (val () (atomic) ((?) <xpr>))))
```

The type "plist" is that of property list. A property list is a list of odd length such that the even numbered elements are atomic.

```
(define plist (type () ((?) <star (atomic) (?)>))
```

We can write the intentions for car as follows.

```
(define car (lambda (((xpr) x))
(; the function car has one argument which is of type (xpr))
(overall ((OLD-X $$x))
        (goal (not !(atom $$x)!)
        (car $$x)
        (thlambda (Y) (thprog (W)
                (assert (eq $$OLD-X $$x)))))))
```

```
(define cdr (lambda (xpr) (((xpr) x))
(overall ((OLD-X $$x))
        (goal (not !(atom $$x)!))
        (cdr $$x)
        (thlambda (Y)
                (assert (eq $$OLD-X $$x))))))
```

The function "identity" which is used below is the identity function.

```
(define cons (lambda (xpr) (((?) x) ((xpr) y))
(overall ((OLD-X $$x) (OLD-Y $$y))
        t
        (cons $$x $$y)
        (thlambda (Z)
                (thprog ((t (now)))
                        (assert (theorem (consequent (u w)
                                (not (descendant $$Z (time $?w
$?u)))
                                (goal (before $$u $$t))))
                (assert (eq $$OLD-X $$x))
                (assert (eq $$OLD-Y $$y))
                (assert (eq !(car $$Z)! $$x))
                (assert (eq !(cdr $$Z)! $$y))
                (assert (not !(atom $$Z)!)))))))
```

Allowing side effects considerably complicates the process of proving intentions. We shall proceed by borrowing a trick from the the Greenblat-Nelson LISP compiler. With each computed expression we will associate the time at which it was computed. The actor (TIME e t) will match an expression e that was computed at time t. The current time will be the value of the the function (NOW). The function (NEXT t) will evaluate to the next time after t. The statement (descendant x y) wil be true only if x can be obtained from a car-cdr chain from y. For example (descendent (car (car x)) x) is true.

```
(define get (lambda (((atomic) p) ((atomic) ind))
(overall ((OLD-P $$p) (OLD-IND $$ind))
        t
        (get $$p $$ind)
        (thlambda (Z)
                (assert (eq $$OLD-P $$p))
                (assert (eq $$OLD-IND $$ind))))))

(define put (lambda (((atomic) p) ((atomic) ind) ((?) value))
(overall ((OLD-IND $$ind) (OLD-VALUE $$value))
        t
        (put $$p $$ind $$value)
        (thlambda (X)
                (assert (eq $$X $$p))
                (assert (eq $$OLD-IND $$ind))
                (assert (eq $$OLD-VALUE $$value))
                (assert (theorem (antecedent (I)
                        (not (= $?I $$ind))
                        (assert (eq !(get $$p $$I)! !(get $$p
$$ind)!)))))
                (assert (theorem (consequent (I)
                        (eq !(get $$p $?I)! !(get $$p $$ind)!)
                        (goal (= $$I $$ind)))))
                (assert (eq $$value !(get $$p $$ind[!)))))))
```

Using the above intentions, we can prove the following intention.

```
(overall (Y)
        t
        (get
                (put $$p $$ind1 $$value)
                $$ind2)
        (thlambda (X)
                (thcond
                        ((goal (= $$ind1 $$ind2))
                        (goal (eq $$X $$value)))
                        ((goal (not (= $$ind1 $$ind2)))
                        (goal (eq $$X !(get $$p $$ind2)!)))))))

(define rplacd (lambda (((xpr) x) ((?) y))
(overall ((OLD-X $$x) (OLD-Y $$y) (T (now)))
        (goal (not !(atom $$x)!))
        (rplacd $$x $$y)
        (thlambda (Z)
                (assert (eq $$Z $$x))
                (assert (eq $$OLD-Y $$y))
                (assert (theorem (consequent (W)
                        (eq (time $?W $$T) (time $?W (next
$$T)))
```

```
                        (goal (not (descendent $$x (time $$W
$$T)))))))   (assert (eq !(car $$OLD-X)! !(car $$x)!))
                (assert (theorem (antecedent (W)
                        (not (descendent $$x (time $?W $$T))))
                        (assert (eq (time $$W $$T) (time $$W
(next $$T)))))))
                (assert (eq !(cdr $$x)! $$y))))))

(define rplaca (lambda (((xpr) x) ((?) y))
 (overall ((OLD-X $$x) (OLD-Y $$y) (T (now)))
        (goal (not !(atom $$x)!))
        (rplaca $$x $$y)
        (thlambda (Z)
                (assert (eq $$Z $$x))
                (assert (eq $$OLD-Y $$y))
                (assert (theorem (consequent (W)
                        (eq (time $?W $$T) (time $?W (next
$$T)))
                        (goal (not (descendent $$x (time $$W
$$T)))))))   (assert (eq !(car $$OLD-X)! !(car $$x)!))
                (assert (theorem (antecedent (W)
                        (not (descendent $$x (time $?W $$T))))
                        (assert (eq (time $$W $$T) (time $$W
(next $$T)))))))
                (assert (eq !(cdr $$OLD-X)! !(cdr $$x)!))
                (assert (eq !(car $$x)! $$y))))))
```

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 declaration 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 $$f))
        t
        $←f
        (thlambda (X Y)
                (assert (eq $$f $$X))
                (assert (= $$Y t)))
        (thlambda (X Y)
                (thcond
                        ((goal (= $$X ())
                        (assert (eq $$f $$OLD-F))
                        (assert (= $$Y ())))
                        ((goal (= $$X t))
                        (assert (eq $$1 $$X))
                        (assert (= $$Y t))))))
```

The value of f changes only if the rest of the match succeeds.

The actor (ENTIRE declaration 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 (kappa ()

(entire ()
        t
        (atomic)
        (thlambda (X Y)
                (thcond
                        ((goal !(atom $$X)!))
                        (assert (= $$Y t)))
                        ((goal (not !(atom $$X)!))
                        (assert (= $$Y ()))))))
        (thlambda (X Y)
                (assert (= $$X $$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))
        (thlambda (X Y)
                (thcond
                        ((goal (proved (subset a b)))
                        (assert (= $$X ()))
                        (assign $:already-proved t)
                        (assert (= $$Y ())))
                        ((goal (not (proved (subset a b))))
                        (assert (proved (subset a b)))
                        (assert (= $$X t))
                        (assert (= $$Y (subset a b))))))
        (thlambda (X Y U V)
                (thcond
                        ((is () $$already-proved)
                        (thcond
                                ((goal (= $$X ())
                                (erase (proved (subset a
b)))))))
                (assert (= $$U $$X))
                (assert (= $$V $$Y))))
```

We would like to show that if we reverse a list twice then we get the original list.

```
(define reverse (lambda (l)
(overall ()
     t
     (intent
        t
        (prog (<?> (u $$l) (v ()))
        again
                (overall ()
                        (goal (= $$v !(reverse !(sub $$l
$$v)!)!)))))
```

```
                (intent
                        (assert (= $$v !(reverse !(sub $$l
$$v)!)!)))
                (cond
                        ((is () $$u)
                        (return $$v)))
                (assign $:v ((l $$u) $$v))
                (assign $:u (rest $$u))
                (go again))
        (thlambda (X)
                (goal (= $$X !(rev $$l)!))
                (goal (= $$l !(reverse $$X)!)))
     (thlambda (X)
                (assert (= $$X !(rev $$l)!))
                (assert (= $$l !(reverse $$X)!))))))

(define sub (lambda (x y)
(overall ()
     t
     (intent
        t
        (cond
                ((is $$x $$y)
                ())
                (t
                ((l $$x) <sub (rest $$x)> $$y)))))
        (thlambda (Z)
                (thcond
                        ((goal (= $$y ()))
                        (goal (= $$Z $$x))))))
     (thlambda (Z)
                (thcond
                        ((goal (= $$y ()))
                        (assert (= $$Z $$x)))))))

(define rev (lambda (l)
(overall ()
     t
     (intent
        t
        (cond
                ((atom $$l)
                $$l)
                (t
                (<rev (rest $$l)> (l $$l))))
        (thlambda (X)
                (goal (= $$X !(reverse $$l)!))
                (goal (= $$l !(reverse $$X)!)))
     (thlambda (X)
                (assert (euql $$X !(reverse $$l)!))
```

```
(assert (= $$l !(reverse $$X)!))))))
```

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 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 ior example). The teacher is duty bound to tell anything that might help the one being taught to understand the properties and structure oi the procedure. We assume that the teacher has a good model oi 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 oi the action oi the procedure for various inputs and enviroments. By "variablization" (the introduction oi identifiers for the constants oi 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 irom examples the procedural abstraction oi protocols. Procedural abstraction can be used to teach oneseli 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 and define the statement (wall $←w $←h) to mean that there is a wall of width $$w and height $$h using the definition (conjunction (((fix) w)) w initial 0 step 1 until (= $$ww $$w)(conjunction (((fix) hh)) hh initial $$h step -1 until (=0 $$hh) (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.

```
(define build-tower
 (consequent
        ((((fix) w h) (<?> (actions ())))
        (brick-at $?w $?h)
                (thcond
                        ((not (hasval? $$h))
                        (assign $$h 0)
                        (go rest))
                        ((= 0 $$h)
                        (go rest)))
                (assign ($←actions) (goal (brick-at $?w (- $$h
1))))
         rest
                (thcond ((proved? (brick-at $?w $?h))
                        (return ())))
                (goal (put-brick-at $?w $?h))
                (goal (check-brick-at $$w $$h))
                (assert (brick-at $$w $$h))
                (return ($$actions !(put-brick-at $$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 ($$ 9) to mean a reference to an identifier whose value is 9; the expression ($: 3 7) means that an identifier with value 3 is assigned the value 7. More precisely, the protocol for (wall 1 2) is

```
(new [1 2]
(new [NO-VALUE NO-VALUE]
(assign ($: NO-VALUE 0) 0)
FALSE: (= ($$ 0) ($$ 1))
SO
(assign ($: NO-VALUE 0) 0)
FALSE: (= ($$ 0) ($$ 2))
     SO
        (put-brick-at ($$ 0) ($$ 0))
        (assign ($: 0 1) (+ ($$ 0) 1))
        FALSE: (= ($$ 1) ($$ 2))
             SO
                (put-brick-at ($$ 0) ($$ 1))
                (assign ($: 1 2) (+ ($$ 1) 1))
                TRUE: (= ($$ 2) ($$ 2))
                     SO
                        (assign ($: 0 1) (+ ($$ 0) 1))
                        TRUE: (= ($$ 1) ($$ 1))
                             SO
                               ()
```

The protocol for (wall 2 1) is

```
(new [2 1]
(new [NO-VALUE  NO-VALUE]
(assign ($: NO-VALUE 0) 0)
FALSE: (= ($$ 0) ($$ 2))
```

```
 SO
(assign ($: NO-VALUE 0) 0) FALSE: (= ($$ 0) ($$ 1))
     SO
        (put-brick-at ($$ 0) ($$ 0))
        (assign ($: 0 1) (+ ($$ 0) 1))
        TRUE: (= ($$ 1) ($$ 1))
             SO
                (assign ($: 0 1) (+ ($$ 0) 1))
                FALSE: (= ($$ 1) ($$ 2))
                     SO
                        (assign ($: 1 0) 0)
                        FALSE: (= ($$ 0) 1)
                             SO
                                (put-brick-at ($$ 1) ($$ 0))
                                (assign ($: 0 1) (+ ($$ 0) 1))
                                TRUE: (= ($$ 1) ($$ 1))
                                     SO
                                        (assign ($: 1 2) (+ ($$
1) 1))
                                        TRUE: (= ($$ 2) ($$ 2))
                                             SO ()

The protocol for (wall 2 2) is
(new [2 1]
        (new [NO-VALUE NO-VALUE]
        (assign ($: NO-VALUE 0) 0)
        FALSE: (= ($$ 0) ($$ 2))
        SO
        (assign ($: NO-VALUE 0) 0)
        FALSE: (= ($$ 0) ($$ 2))
        SO
        (put-brick-at ($$ 0) ($$ 0))
        (assign ($: 0 1) (+ ($$ 0) 1))
        FALSE: (= ($$ 1) ($$ 2))
        SO
        (put-brick-at ($$ 0) ($$ 1))
        (assign ($: 1 2) (+ ($$ 1) 1))
        TRUE: (= ($$ 2) ($$ 2))
        SO
        (assign ($: 0 1) (+ ($$ 0) 1))
        FALSE: (= ($$ 1) ($$ 2))
        SO
        (assign ($: 2 0) 0)
        FALSE: (= ($$ 0) ($$ 2))
        SO
        (put-brick-at 1 0)
        (assign ($: 0 1) (+ ($$ 0) 1))
        FALSE: (= ($$ 1) ($$ 2))
        SO
        (put-brick-at ($$ 1) ($$ 1))
```

```
        (assign ($: 1 2) (+ ($$ 1) 1))
        (assign ($: 1 2) (+ ($$ 1) 1))
        TRUE: (= ($$ 2) ($$ 2))
        SO
        ()
```

by introducing identifiers for the constants the protocols can be arranged in a tree as follows:

```
new [w h]
new [ww=NO-VALUE; hh=NO-VALUE]
(assign $:ww 0)
i1 (= $$ww $$w)
then
()
 else (assign $:hh 0) if (= $$hh $$h)
     then
        (assign $:ww (+ $$ww 1))
        if (= $$ww $$w)
             then
                ()
             else
                (assign $:hh 0)
                if (= $$hh 0)
                     then
                        (assign $:ww (+ $$ww 1))
                                if (= $$ww $$w)
                                     then
                                        ()
                                     else...
                     else...
     else
        (put-brick-at $$ww $$hh))
        (assign $:hh (+ $$hh 1))
        if (= $$hh $$h)
             then
                (assign $:ww (+ $$ww 1))
                i1 (= $$ww 1)
                     then
                        ()
                     else
                        (assign $:hh 0)
                        if (= $$hh $$h)
                             then
                                (put-brick-at $$ww $$hh)
                                (assign $:hh (+ $$hh 1))
                                        if (= $$hh $$h)
                                             then
                                                (assign $:ww (+
$$ww 1))
```

```
                                                    if (= $$ww $$w)
                                                         then ()
                                                         else...
                                                else...
                        else...
        else
           (assign $:hh (+ $$hh 1))
           if (= $$hh $$h)
                then
                   (assign $:ww (+ $$ww 1))
                   if (= $$ww $$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 con 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 (lambda (((fix) w) ((fix) h))
(overall ()
     (thand
        (goal (greaterp $$w 0))
        (goal (greaterp $$w 0)))
     (intent
        (thand
                (assert (greaterp $$w 0))
                (assert (greaterp $$w 0)))
        (prog
                (((fix) ww hh))
                (assign $:ww 0)
                (goal (= $$ww 0))
        column
                (overall ()
                        (goal (wall $$ww $$h))
                (intent
                        (assert (wall $$ww $$h))
                (cond
                        ((= $$ww $$w)
                        (intent (wall $$w $$h))
                        (return ())))
                (assign $:hh 0)
                (intent (goal (= $$hh 0)))
        height
                (overall ()
                        (thand
                                (goal (wall $$ww $$h))
                                (goal (column $$ww $$hh))))
                (intent
                        (thand
                                (assert (wall $$ww $$h))
                                (assert (column $$ww $$hh))))
                (cond
                        ((= $$hh $$h)
                        (assign $:ww (+ $$ww 1))
                        (go column))))
                (intent (goal (support-for $$ww  $$hh)))
                (put-brick-at $$ww $$hh)
                (intent (goal (brick-at $$ww $$hh)))
                (assign $:hh (+ $$hh 1))
                (go height))
        (thlambda (X)
                (goal (wall $$w $$h)))
     (thlambda (X)
        (assert (wall $$w $$h)))))

(define  check-wall
     (consequent
```

```
        (w' w h' h)
        (wall $?w' $?h')
        (thcond
                ((thor
                        (goal (= $?h' 0))
                        (goal (= $?w' 0))))
                ((is !(+ $?h 1)! $$h')
                        (goal (wall $?w' $$h)))
                        (goal (column $?w' $?')))
                ((is !(+ $?w 1)! $$w')
                        (goal (wall $?w $$h')))
                        (goal (column $?w' $?h')))
                (t
                        (fail theorem)))))

(define check-column
    (consequent
        (w h h')
        (column $?w $?h')
        (thcond
                ((goal (= $?h' 0)))
                ((is !(+ $?h 1)! $$h')
                        (goal (column $?w $?h)))
                (t
                        (fail theorem)))))

(define check-support
    (consequent
        (w h)
        (support-for $?w $?h)
        (thcond
                ((goal (= $?h 0)))
                ((goal (column $$ww $$hh)))
                (t (fail theorem)))))

(define put-brick-at
    (thlambda (w h)
        (overall ()
                (goal (support-for $$w $$h))
                (put-brick-at $$w $$h)
                (assert (brick-at $$w $$h)))))
```

The function compile-build has been simplified by transforming the recursive function that correspond to the tags column and height into loops. 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 $$ww 0)
                (wall (subl $$ww) $$hh)))
```

is evaluated immediately after (go 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 $$w 0)) (wall (subl $$ww) $$hh)). Of course using protocols for the purpose o1 rejecting false subgoals does not help us to eliminate those that are true but unprovable.

7.2.2.4.2 Reversing a List at All Levels

Consider the following protocols for a procedure r:

```
(new [a]
TRUE: (is (atomic) ($$ a))
SO ($$ a)

thus (r a) is a


(new [(n)]
FALSE: (is (atomic) ($$ (n)))
     SO
        (
        <new [(rest ($$ (n)))]
        TRUE: (is (atomic) ($$ ()))
             SO ($$ ())>
        (new [(l ($$ (n)))]
        TRUE: (is (atomic) ($$ n))
             SO ($$ n)))

thus (r (n)) is (n)


(new [(a b)]
FALSE: (is (atomic) ($$ (a b)))
     SO
        (
        <new [(rest ($$ (a b)))]
        FALSE: (is (atomic) ($$ (b)))
             SO
                (
                <(new [(rest ($$ (b)))]
                TRUE: (is (atomic) ($$ ()))
                     SO ($$ ())>
```

```
                (new [(1 ($$ (b)))]
                TRUE: (is (atomic) ($$ b))
                     SO ($$ b)))>
          (new [(1 ($$ (a b)))]
          TRUE: (is (atomic) ($$ a)))
               SO ($$ a)))
```

thus (r (a b)) is (b a)

```
(new [((a))]
FALSE: (is (atomic) ($$ ((a))))
     SO
        (
        <(new [(rest ($$ ((a))))]
        TRUE: (is (atomic) ($$ ()))
             SO ()>
        (new [(1 ($$ ((a))))]
        FALSE: (is (atomic) ($$ (a)))
             SO
                (
                <(new [(rest ($$ (a)))]
                TRUE: (is (atomic) ($$ ()))
                     SO ())>
                (new [(1 ($$ (a)))]
                TRUE: (is (atomic) ($$ a))
                     SO ($$ a)))))
```

thus (r ((a))) is ((a))

We obtain the following protocol tree:

```
(new [x1]
i1 (is (atomic) $$x1)
     then $$x1
     else
        (
        <new [x2 (rest $$x1)]
        if (is (atomic) $$x2)
             then $$x2
             else
                (
                <new [x3 (rest $$x2)]
                if (is (atomic) $$x3)
                     then $$x3
                     else...>
                (new [x4 (1 $$x2)]
                i1 (is (atomic) $$x4)
                     then $$x4
```

```
                    else...))>
        (new [x5 (1 $$x1)]
        if (is (atomic) $$x5)
             then $$x5
             else
                (
                <new [x6 (rest $$x5)]
                if (is (atomic) $$x6)
                     then $$x6
                     else...>
                (new [x7 (1 $$x5)]
                i1 (is (atomic) $$x7)
                     then $$x7
                     else...)))
```

By identifying indistinguishable nodes we obtain:

```
(define super-reverse (lambda (x)
        (cond
                ((is (atomic) $$x) $$x)
                (t (<super-reverse (rest $$x)> (super-reverse (1
$$x)))))))
```

Consider the following set of protocols:

7.2..2.4.3 Finding the Description of a Stick

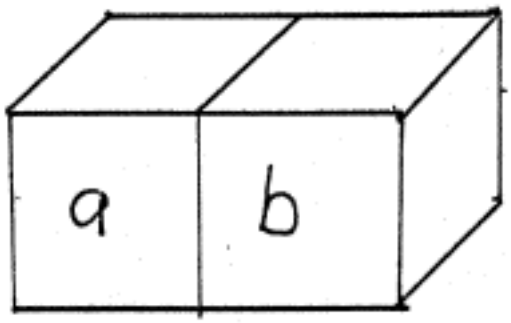
Suppose that we have the following data base:

```
(block a)

(block b)

(glued a b)
```

The above data base represents a stick on the the basis of the following protocol:

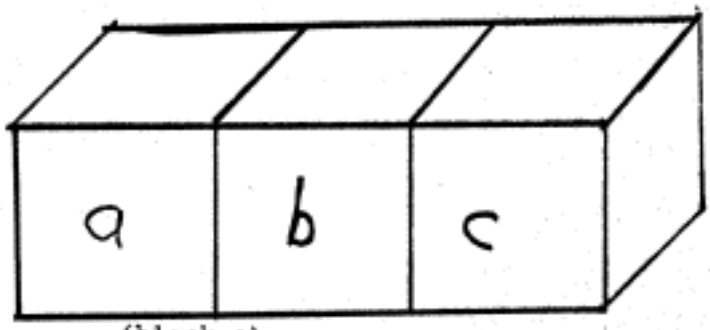
```
(goal (stick a b))
        (new NO-VALUE NO-VALUE NO-VALUE] (; we have three new
identifiers that do not have values)
        consequent: (stick ($? NO-VALUE a) ($? NO-VALUE b))
        thcond
                (proved? (glued ($? a) ($? b)))
```

Figure 1.



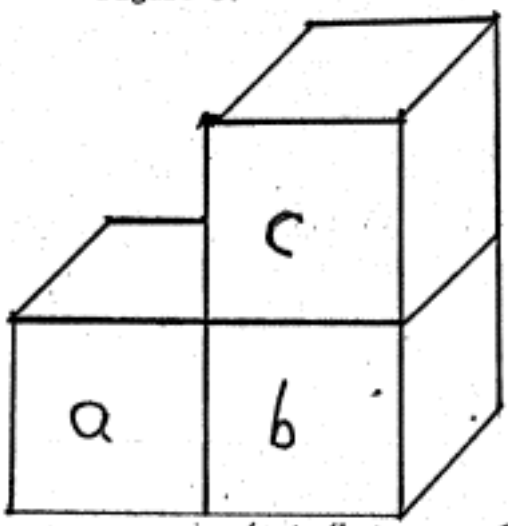
(block a)
(block b)
(glued a b)

Figure 2.



(block a)
(block b)
(block c)
(glued a b)
(glued b c)
(between a b c)

Figure 3.



(block a)
(block b)
(block c)
(glued a b)
(glued b c)
(not (between a b c))

```
                (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 NO-VALUE NO-VALUE NO-VALUE]
        consequent:  (stick ($? a) ($? c))
        thcond
                (proved? (glued ($? a) ($? c)))
                fail
        (proved (block ($? a)))
        (goal (glued ($$ a) ($- NO-VALUE b)))
        (proved? (between ($$ a) ($$ b) ($? c)))
        (goal (stick ($$ b) ($$ c)))
                [new NO-VALUE NO-VALUE NO-VALUE]
                consequent:  (stick ($? b) ($? c))
                thcond
                        (proved (glued ($? b) ($? c)))
                        (return t)
```

By variabalization we obtain the following protocol tree:

```
(goal (stick u v))
        [new x y z]
        consequent: (stick $?x $?z)
        (thcond
                ((goal (glued $?x $?z))
                (return t)))
        (proved? (block $?x))
        (goal (glued $$x $-y))
        (proved? (between $$x $$y $?z))
        (goal (stick $$y $$z))
                [new x1 y1 z1]
                consequent:  (stick $?x1 $?z1)
                (thcond
                        ((goal (glued $?x1 $?z1))
                        (return t)))
                (proved? (block $?x1))
                (goal (glued $$x1 $-y1))
                (proved? (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-description (consequent
        (x y z)
        (stick $?x $?z)
                (thcond
                        ((goal (glued  $?x $?z))
                        (return t)))
                (proved? (block $?x))
                (goal (glued $$x $$y))
                (proved? (between $$x $$y $?z))
                (goal (stick $$y $$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 (lambda (iix) (((fix) n))
        (cond (iix) ((or (P n) (P (S n))) 0)
                (t (A (i (S n)) (i (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:

```
ii (or (P (S^0 n)) (P (S^1 n)))
     then 0
     else
        (A
                ii (or (P (S^1 n)) (P (S^2 n)))
                     then 0
                     else
                        (A
                                ii (or (P (S^2 n)) (P (S^3 n)))
                                     then 0
                                     else...
```

```
                                    if (or (P (S^3 n)) (P (S^4 n)))
                                        then 0
                                        else...)
                    if (or (P (S^2 n)) (P (S^3 n))))
                        then 0
                        else
                            (A
                                    if (or (P (S^3 n)) (P (S^4 n)))
                                        then 0
                                        else...
                                    if (or (P (S^4 n))  (P (S^5 n)))
                                        then 0
                                        else...))
```

By procedural abstraction we can obtain a function f1 which is equivalent to 1. The function is obtained by identifying some of the nodes that are not on the same branch of the protocol tree. The type (fix) below is the type of fixed point numbers. The type (loc (fix)) is the type of where the location of a fixed point number can be stored.

```
(define f1 (lambda (fix) (((fix) n))
        (prog (((loc (fix)) m))
                (f2 n m))))

(define f2 (lambda (fix) (n ((loc (fix)) m))
        (cond
             ((or (P n) (P (S n))
                (; in this special case smash the location m to
contain 0)
                (assign (smash $$m) 0))
             (t
                (prog (fix) (((loc (fix)) 1))
                        (assign (smash $$m) (f2 $$n $$1))(; set
the contents of m to be (f2 $$n $$1))
                        (; the value of f2 is the function A
applied to the contents of m and the contents of 1)
                        (A
                                (in $$m)
                                (in $$1)))))))
```

Another approach is to use some of the theory of recursive

schemas. The function 1 defined above is schematically equivalent to the function f1 defined below

```
(define ff (lambda (fix) (((fix) n))
        (for (fix) (((fix) (x 0) (y 0))))
                        ((test (P $$n) (return $$x))
                        (step (assign $:n (S $$n))))
        (assign ![$:x $:y]! ![(A $$x $$y) $$x]!)
                (;the previous statement is just a tricky way to
simultaneously accomplish (assign $:x (A $$x $$y)) and (assign
$:y $$x)))))
```

Note that (fib n) the nth Fibonacci number can be defined as follows

```
(define fib (lambda (fix) (((fix) n))
        (cond (fix)
                ((or (is 1 $$n) (is 2 $$n)) 1)
        (t (+ (fib (- $$n 1) (- $$n 2)))))))
```

Using the interpretation that 0 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 suifer from the inefficiency of double interpretation. Instead oi 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 oi protocols along the same lines for actors. For example if we obtain the iollowing actor protocol

```
((i1
        (())
        ((atomic))
        ((if
                (())
                ((atomic))
                ((i1
                        (())
                        ((atomic)))
                <if
                        (())>)
        <i1
```

```
((()
((atomic))
((if
        (())
        ((atomic))
        ((if
                ((())
        <if
                (())
                ((atomic))>)))))
```

Then by identifying equivalent nodes we obtain the actor expr where

```
(define expr (kappa ()
        (if
                (())
                ((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.

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 "act". 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 (REVERSEP x y) which means that y is the reverse of x. For example (reversep aa aa) and (reversep (1 2 (3 4)) ((3 4) 2 1)) are true. We shall use !< and >! as meta angle brackets for < and > respectivly. As before !(and)! are the meta braces for (and).

```
(define th69 (consequent
        (a b c)
        (reversep $?a $?b)
        (thcond
                ((hasval? a)
                (thcond
                        ((goal (atom $$a))
                        (; if a is an atom then b should be
equal to a)
                        (goal (= $$a $?b)))
                        (t
                        (goal (not (atom $$a)))
```

```
                        (goal (reversep !(rest $$a)! $-c))
                        (; otherwise let c bethe reverse o1 the
rest o1 a)
                        (goal (= (!<identity $$c>! !(1 $$a)!)
$?b)))))
                (t (gen1ail)))))
```

We would like to find a 1unction reverse such that (reversep 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 1or linear induction applied to the 1unction reverse is:

```
(de1ine reverse (lambda (x)
        !(cond
                (!(atom $$x)!
                (temprog (Y)
                        (assert (atom $$x))
                        (goal (reversep $$x $-Y))
                        (; find a Y which is the reverse o1 the
atom "$$x" and return it as value)
                        $$Y))
                (t
                (temprog (Y)
                        (assert (not (atom $$x)))
                        (assert (reversep
                                !(rest $$x)!
                                !(reverse !(rest $$x)!)!))
                        (goal (reversep $$x $-Y))
                        $$Y)))))
```

The above expression evaluates to the following definition:

```
(de1ine reverse (lambda (x)
        (cond
                ((atom $$x) $$x)
                (t (<identity (reverse (rest $$x))> (1 $$x))))))
```

7.2.4. Comparison of the Methods

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. 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 way 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 we have made to the language are first steps toward realizing the Principle of Procedural Embedding.

7.3 Current Problems and Future Work

BIBLIOGRAPHY

Black, F., 1964. A Deductive Question Answering System, doctoral dissertation, Harvard University, Cambridge, Mass.

Cooper, D. C., The Equivalence of Certain Computations, The Computer Journal, Vol. 9, no. 1.

Floyd, R. W., Assigning meanings to Programs, Proceedings Of Symposia in Applied Mathematics. Volume XIX.

Floyd, R. W., Nondeterministic Algorithms, JACM. Oct. 1967.

Green, C. C. and Raphael B. Research on Intelligent Question-answering System. May 1967.

Green C. C. (with Bob Yates), 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., Functional Abstraction in LISP and PLANNER, Artificial Intelligence Memo 151, Massachusetts Institute of

Technology (project MAC),

Kaplan, D. M., Correctness of a Compiler for ALGOL-like Programs. Stanford A.I. Memo No. 48.

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. 1962. Lisp 1.5 Programmer's Manual, M. I. T. Press.

McCarthy, J. and Hayes, P., Some Philosophical Problems form the Standpoint of Artificial Intelligence. Stanford A. I. Memo 73.

Newell A., Studies in Problem Solving: Subject 3 on the Crypt-arithmetic Task Donald + Gerald = Robert.

Newell, A., Shaw, J. C., and Simon, H. A., 1959. Report on a General Problem-solving Program, Proceedings of the International Conference on Information Processing, Paris: UNESCO House, pp. 256-264.

Manna, Z., Termination of Algorithms. Ph. D. thesis, Carnegie.

Rovner, Paul D. LEAP Users Manual. Lincoln Laboratory Technical Memorandum No. 231-0009.

Slagle, J., 1965. Experiments with a Deductive Question-answering Program, Communications of the Association for

Computing Machinery, December, 8:792-798.

Waldinger and Lee, PROW: A Step Toward Automatic Program Writing. Proc. IJCAI.