SRI International

QA4: A PROCEDURAL CALCULUS FOR INTUITIVE REASONING

Technical Note 73

November 1972

By: Johns F. Rulifson
Jan A. Derksen
Richard J. Waldinger

Artificial Intelligence Center

SRI Project 8721

The research reported herein was sponsored by the National Aeronautics and Space Administration under Contract NASW-2086.

333 Ravenswood Ave. • Menlo Park, CA 94025
(415) 326-6200 • TWX: 910-373-2046 • Telex: 334-486

ABSTRACT

This report presents a language, called QA4, designed to facilitate the construction of problem-solving systems used for robot planning, theorem proving, and automatic program synthesis and verification. QA4 integrates an omega-order logic language with canonical composition, associative retrieval, and pattern matching of expressions; process structure programming; goal-directed searching; and demons. Thus it provides many useful programming aids. More importantly, however, it provides a semantic framework for common sense reasoning about these problem domains. The interpreter for the language is extraordinarily general, and is therefore an adaptable tool for developing the specialized techniques of intuitive, symbolic reasoning used by the intelligent systems.

Chapter Two is a primer for the QA4 language. It informally presents the language through the use of examples. Most of the unusual or complicated features of the language are not discussed. The Chapter concludes with a presentation of a small robot planning systems that uses only the language features presented in the Chapter. Chapter Three presents a series of examples chosen to illustrate solutions to automatic programming problems. The QA4 programs used in Chapter Three rely on language features not presented in the primer. They are, however, explained as they occur. These programs illustrate most of the

programming concepts just discussed. Chapter Four is a complete reference guide to the language. It provides the semantics of all the features of the language together with many implementation notes and design rational. Chapter Five discusses extensions to the language that will probably be done during the next year.

ACKNOWLEDGEMENTS

The QA4 project has not been carried out in isolation; it has been performed in the fertile environment of Stanford Research Institute's Artificial Intelligence Center and in the artificial intelligence community in general. Although it is difficult to give credit to all the individuals who have contributed to the QA4 effort, the following is a partial list of the principle contributors.

Cordell Green was the first to see the need for a QA4; he and Robert Yates supplied the initial conceptual framework. Charles Rosen was the manager of the SRI Artificial Intelligence Group at this time; his openness and enthusiasm for new ideas has been a continual source of encouragement.

During its formative period, the project benefitted from discussions with Peter Deutsch, Alan Kay and Erik Sandewall. Furthermore, the name "bag" was supplied by Peter Deutsch. Erik Sandewall invented the table illustrating the meanings of the variable prefixes. Alan Kay helped organize a conference in a beach house at Pajaro Dunes.

Carl Hewitt, who attended the Pajaro Dunes conference, taught us much through the example of his language PLANNER; his advice and criticisms heavily influenced the design of QA4. We were further impressed and inspired by Terry Winograd's use of a PLANNER subset in the implementation of his BLOCKS program.

Richard Fikes helped us formulate our ideas about goal structures, backtracking, and the use of processes in robot planning.

Bertram Raphael, the present Director of the Artificial Intelligence Center, has given us detailed and thorough criticism of the QA4 conception, implementation, and manuscript drafts.

Richard Fikes, Peter Hart and Nils Nilsson have collaborated with us on the application of QA4 to robot problem solving. All our work on robot planning has been based on the STRIPS model and formulation.

Irene Grief helped us debug QA4 by writing portions of a program verifier using an early version of the system.

Ann Robinson, Steven Crocker, Larry Tesler, Bruce Anderson, and Richard Fikes have given us readings and criticisms of this manuscript. Ann Robinson did a thorough and insightful reading of the proofs.

Thanks are due to Cordell Green for serving as Johns Rulifson's thesis advisor, and to Jerome Feldman and Edward Feigenbaum for serving on his committee.

This work has been supported at SRI by NASA and ARPA under Contracts NAS12-2221, NASW-2164 and NASW-2086. For the past two years the work has been supported primarily by NASA under Contract NASW-2086.

Samuel Rosenfeld, our contract monitor at NASA, has given us exceptional support, encouragement, and money.

We appreciated Kathy Spence's cheerfulness in typing several versions of this 400 page document, and correcting inconsistencies in our notation.

TABLE OF CONTENTS

ABSTRACT iii

ACKNOWLEDGEMENTS v

TABLE OF CONTENTS vii

CHAPTER ONE--PROGRAMMING CONCEPTS 1

I PROJECT DEVELOPMENT 1

A. Introduction 1
B. Project Origin 1
C. The Language 2
D. The Problem Domain 3

II EXPRESSIONS 5

A. Motivation 5
B. Data Structures 5
C. Example 6
D. Composition 8
E. Pattern Matching 9

III CONTROL 11

A. Context 11
B. Motivation 13
C. Indecision 15
D. Iteration 16

IV ORGANIZATION 19

A. Review of Goals 19
B. Goals 19
C. Choosing Solution Programs 22
D. Models and WHENs 23

V CURRENT STATUS 27

CHAPTER TWO--A PRIMER 29

I EXPRESSIONS 29

A. Introduction 29
B. The Primitive Expressions 29

II BUILT-IN FUNCTIONS 35

A. Logical Connectives 35
B. Arithmetic Functions 37
C. Structural Functions 40

III PATTERN MATCHING 43

A. Introduction 43
B. Patterns 45
C. The Fragment Variable Applied to Construction . 51
D. Multiple Matches 51
E. Subpatterns 55

IV EVALUATION AND INSTANTIATION 57

V THE NET: ASSOCIATION INFORMATION WITH EXPRESSIONS . 59

A. Internal Format of QA4 Expressions 59
B. Entering a New Expression in the System 60
C. Advantages of the Net Storage Mechanism 61

VI PROGRAM CONTROL 67

A. Overview 67
B. Backtracking 67
C. The Goal Mechanism 69
D. Failure 72
E. Conditional Statements 74
F. The PROG Feature 78

VII THE CONTEXT MECHANISM 81

A. Introduction 81
B. Creating a Context 82
C. Builting a Tree of Contexts 82
D. Creating a Context by Referring to Foregoing Nodes in a Context Tree 84
E. An Example 84

VIII A ROBOT SYSTEM 91

A. Introduction 91
B. The Robot Problems 92

IX THE SOLUTION OF THE PROBLEM OF TURNING ON A LIGHT . . 95

A. The Framework 95
B. The STRIPS Representation 96
C. The QA4 Representation 98
D. Design Philosophy Revisited 104
E. Other Features and Applications 106

CHAPTER THREE--PLAN SYNTHESIZERS 109

I INTRODUCTION 109

II CONDITIONAL PLANS 111

A. Predicates and Actions 111
B. Goal Satisfaction 112
C. MOVIE and BEACH 113
D. ALTPLAN 114
E. (GOAL $HAVEFUN HAVEFUN) 116

III INFORMATION GATHERING 121

A. Operators 121
B. TESTABLE 122
C. (GOAL $DO (OPEN DOOR)) 124

IV LOOPS 129

A. When To versus How To 129
B. LOOPPLAN 129
C. (GOAL $DO (FA ...)) 130

V CONSTRAINTS 133

A. "Thou shalt not ..." Problems 133
B. Four-Step Operators 133
C. Trying Out the Final State 134
D. Remaining Planner Parts 135
E. (GOAL $DO (INROOM BOX ROOM2)) 137

VI COORDINATED PLANS 145

A. A Shopping Problem 145
B. The Shopping List 149
C. Sorting the List 152
D. Process Control Programs 155

CHAPTER FOUR--THE LANGUAGE 165

I RUNNING THE SYSTEM 165

A. Warnings 165
B. Loading the System 165
C. Talking to QA4 165
D. Establishing a Program 166
E. Commands to Establish Models 167

II PRIMITIVE EXPRESSIONS

A. Types and Formats 169
B. Identifiers 171
C. Tuples, Sets, Bags, and Numbers 171
D. Contexts, Processes, and Semaphores 174
E. Applications 174
F. Bound Variable Expressions 175
G. Statements 184

III PRIMITIVE DATA OPERATIONS 187

A. Logical Operators 187
B. Structural Operations 189
C. Arithmetic Operations 191
D. Constructors 192
E. Syntactic Information 195
F. Decomposition: (NTH t n) 195
G. * Semaphores 196

IV PATTERNS 199

A. Motivation 199
B. Constants 200
C. Variables 201

	D. Instantiation	208
	E. Extended Constructions	209
	F. Internal Representation	213
V	PROPERTIES	215
	A. General Statements	215
	B. Macro Statements	217
	C. Equivalence Relations	218
	D. Special Context and Recommendation Options	228
VI	QUERY STATEMENTS	231
	A. Motivation	231
	B. INSTANCES	231
	C. EXISTS	233
VII	CONTEXTS	235
	A. Canonical Representations	235
	B. Bindings of Properties	235
	C. User Contexts	237
	D. CONTEXT Statement	239
	E. Summary	239
VIII	STANDARD CONTROL STATEMENTS	241
	A. LIST Statement	241
	B. Conditionals	241
	C. Programs	247
	D. Failure	249
IX	WHEN STATEMENTS	251
	A. Motivation	251
	B. Form	252
X	GOAL STATEMENTS	259
	A. Form	259
	B. Example	259
	C. Order and Advice	262

XI PROCESSES 263

A. RESUME Statement 263
B. INCARNATE Statement 266
C. CONNECT Statement 267
D. WAIT Statement 268

XII ITERATION STATEMENTS 269

A. REPEAT Statement 269
B. FIND Statement 272

CHAPTER FIVE--EXPECTATIONS AND REFLECTIONS 275

I PROJECTS 275

A. Imhotep 275
B. Other Projects 278

II EXTENSIONS 281

A. Lamarckian Evolution 281
B. Searching for Goal Solutions 283
C. Transitive Relations 284
D. Efficiencies 285

III TRENDS IN QA4 PROBLEM SOLVERS 289

A. Backtracking 289
B. Efficiency 290
C. Procedural and Declarative Language Development . 291
D. Ad Hoc or General 292

Appendix I--LISTING OF THE ROBOT SYSTEM 293

Appendix II--THE DISCRIMINATION NET 301

I BACKGROUND 301

A. Canonical Forms 301
B. Basic Mechanisms 303

II FIXED RETRIEVAL, REORDERING, and RENAMING 305

A. Coordinate Indexing 305
B. Reordering 307
C. Bound Variable Renaming 308

III HEURISTIC RETRIEVAL WITHOUT RENAMING 311

A. The Heuristic Technique 311
B. The Discrimination Net 312
C. Construction Search 313
D. Associative Search 314

IV SUMMARY 315

A. Storage Consumption 315
B. Time 316
C. Disadvantages 316

Appendix III--CONTEXTS 319

I BINDINGS 319

A. Properties 319
B. Dynamic Context 319
C. Backtracking Context 322
D. Benefits of Dispensed State 324

II ALGORITHMS 327

A. Terminology 327
B. Internal Expression Form 327
C. Retrieval 328
D. Storage 329

III EXAMPLES 331

A. Introduction 331
B. Function Calls 331
C. Cooperating Processes 341
D. Parallel Processes 351

IV IMPLEMENTATION 355

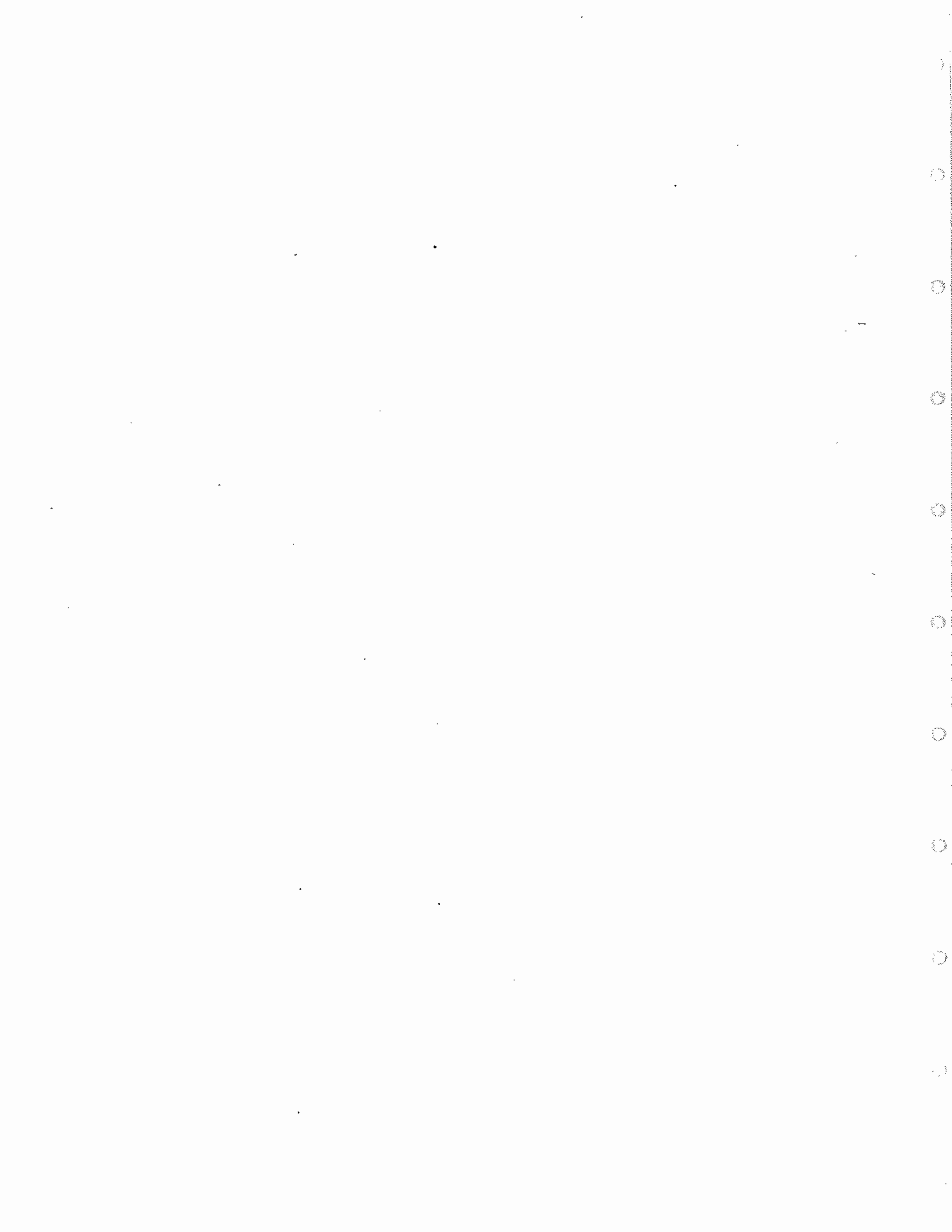
V SUMMARY 357

A. Space and Garbage Collection 357
B. Binding Retrieval Time 357
C. Versatility 358

REFERENCES 361

Chapter One

PROGRAMMING CONCEPTS



CHAPTER ONE--PROGRAMMING CONCEPTS

I PROJECT DEVELOPMENT

A. Introduction

This report presents a language, called QA4, designed to facilitate the construction of problem-solving systems used for robot planning, theorem proving, and automatic program synthesis and verification. QA4 integrates an omega-order logic language with canonical composition, associative retrieval, and pattern matching of expressions; process structure programming; goal-directed searching; and demons. Thus it provides a semantic framework for common sense reasoning about these problem domains. The interpreter for the language is extraordinarily general, and is therefore an adaptable tool for developing the specialized techniques of symbolic reasoning used by the intelligent systems.

This work was begun as part of a general program of research in artificial intelligence supported at Stanford Research Institute by NASA and ARPA under Contracts NAS12-2221 and NASW-2164. For the past two years, the work has been supported primarily by NASA under Contract NASW-2086.

B. Project Origin

QA4 was started by Cordell Green and Robert Yates at SRI just after Green finished his Ph.D. thesis at Stanford University in 1969. His

thesis was on the use of resolution-based theorem-proving systems as a means to automatic Question Answering. (Hence the mnemonic QA.) Their system was named QA3 (Green 1969).* It did, and still does, prove theorems in the first-order calculus using resolution. This system, in fact, is the basis of the SRI ZORBA (Kling 1971) and STRIPS (Fikes 1971) projects.

Green was bothered, however, by the difficulty of trying to use problem-oriented semantic and pragmatic information to guide the theorem prover. Resolution theorem-proving systems are well adapted to syntactic heuristics such as unit preference. They may also be adapted to heuristics that are tied to the deduction mechanism, such as ancestry filter. It was very hard, and sometimes impossible, to use the semantics of the actual problem at hand.

C. The Language

Thus the original goal of the QA4 project was to write a theorem prover for automatic question answering. The formal language was to be far more natural than first-order predicate calculus. This theorem prover was to perform expression transformations on concise expressions in such a way that it produced proofs with a natural style--the kind that we would accept as being intuitive and obviously dominated by the semantics of the problem. As we began to write such a theorem prover, however, we were continually confronted with the restrictions of LISP

---

*References are listed at the end of this report.

(McCarthy 1962, 1963). We wanted our program to plan and reason in a common sense way (McCarthy 1958). Thus we felt that the first step was to produce some theorem-proving protocols that looked intuitive, and to be sure that these protocols could be guided by natural strategies--the kind of advice you would give students. Then we should design a system that could take such strategies and attempt to execute them. When the strategies fail, we want easy, accessible methods of adding more advice and program reorganization. We felt that the project would proceed iteratively--we would start a language and a theorem prover simultaneously, and let each guide the development of the other.

This Chapter explains the attitudes that have evolved about the process of program construction. We will discuss the facets of the QA4 language that permit us to specify our problem solver in the vagueness in which it is conceived and to refine it into an intelligent program. But most importantly, we want to program without losing our way simply because we had to express our thoughts in a language with strict rules about evaluation such as ALGOL or LISP.

D. The Problem Domain

Both the search space and the solutions for the problems we are concerned with are small. Consider an example program verification problem. Using a resolution-proof method, there are over 200 individual, necessary steps in the proof of the program's correctness. By using extended omega-order logic (Robinson 1969) and simplification methods,

the proof can be reduced to about 20 steps. Of these, 15 are obvious deductions (e.g., from A&B deduce both A and B). The remaining five steps require ingenious instantiations and use of induction. We expect a QA4 program verifier to have many special rules and detailed advice on their use. It should produce the 20-step proof with little or no wasted effort. Thus the emphasis in our language design is to permit the specification of many high-level rules and strategies. We anticipate that individual strategy steps may be time-consuming, but that each step is valuable.

II EXPRESSIONS

A. Motivation

Remember that our original goal was to write a theorem prover that proceeded according to pragmatic, intuitive protocols. Seemingly simple axioms and inference rules normally presented in a mixed English-logic language in textbooks often become lengthy, complex formulas when converted to the notation of either first-order predicate calculus or standard programming languages. To even describe our protocols, we needed a concise, natural syntax for algebraic expressions. At the same time, the definitions should mirror the semantics of the primitive operators. Most formal language definitions are guided more by the syntactic properties of the symbols than by the semantics of the operators they stand for. Because of this, these definitions lead to endless applications of transitivity, associativity, and equality inference rules. To prove that X + Y = Y + X, for example, should not require any substeps--it should be immediately obvious to even the simplest theorem-proving programs.

B. Data Structures

Declarative statements in the language have convenient extensions of omega-order calculus such as sets, special quantifiers, and extended primitive operators. The three basic data structures are tuples, bags, and sets:

* Tuples

  Tuples are ordered lists; our notation is (TUPLE 1 2 3)

* Bags

  Bags are unordered tuples. That is, a bag is a collection of unordered elements, and the elements may be duplicated. Our notation is (BAG 3 1 2 1)

* Sets

  Sets are unordered collections of elements, and without duplication. Our notation is (SET 2 3 1). During the construction of sets, either during input or while a program is running and building a set, duplicate elements are automatically removed. Even during user input, multiple occurrences of a variable are reduced to a single occurrence.

C. Example

1. PLUS

Our definition of PLUS illustrates the use of these structures. PLUS is associative and commutative, so it may take a bag as its argument. A user may type (PLUS 1 2 1) as a line of a program. Internally, QA4 uses a prefix representation; thus that line means (PLUS (BAG 1 1 2)). PLUS may not take a set as an argument, for then we would have (PLUS 1 1) = (PLUS (SET 1)) = 1.

2. EQUAL

Our definition of EQUAL is another example of the use of these structures. EQUAL is not merely associative and commutative, but it is also an equivalence relation. Therefore, it may take a set as an argument. X = Y = Z means (EQUAL (SET X Y Z)). During the evaluation of this expression, the set is evaluated by first evaluating the members and then collecting the resulting values into a set. The function EQUAL is then applied to that set. EQUAL is TRUE if and only if the value of its argument has a single member. Thus if X and Y were true, and Z was FALSE, then the value of (SET X Y Z) would be the two-element set (SET TRUE FALSE), and therefore the value of (EQUAL (SET X Y Z)) would be FALSE.

3. X + Y = Y + X

Within QA4, the example we considered earlier--X + Y = Y + X--means, by definition, EQUAL applied to a set. That set has the single member +[X,Y].

typed in	(EQUAL (PLUS X Y)(PLUS Y X)
initial internal form	(EQUAL (SET (PLUS (BAG X Y)) (PLUS (BAG Y X))))
reduced standard form	(EQUAL (SET (PLUS (BAG X Y))))

Thus either during program interpretation or expression simplification within a theorem prover, the value of the expression is obviously TRUE.

D. Composition

1. Canonical Forms

As expressions are composed, they are converted to a canonical form so that semantic and pragmatic properties attached to them can be associated automatically with all equivalent expressions. Composition takes place whenever a particular data structure is constructed by a program. The canonical composition provides for continual syntactic simplification, a feature vital for program verification and theorem proving.

2. Example

For example, the process of interpreting the statement

(SETQ ←X (SET RED BLUE GREEN))

not only assigns X to be a set, but also composes a canonical representation of the set. Variables are always identified by a prefix character. ← means the variable is to receive a value, $ means the variable must have a value. The composition process ensures that if the datum described by the expression (in this case, a set) has ever been previously constructed, the original value is used and no new equal but different structure is constructed.

3. Bound Variables

This identification of equivalence is even made between expressions that include bound variables. Thus the functions

```
(LAMBDA (TUPLE ←X ←Y)(TIMES (PLUS $X $Y)(PLUS $Y 1)))
```

and

```
(LAMBDA (TUPLE ←U ←V)(TIMES (PLUS 1 $V)(PLUS $V $U)))
```

will both be converted to the same internal canonical form, and information known about one is always available to strategies that may deal with the other.

E. Pattern Matching

1. How It Integrates

The use of canonical representation together with definitions that reflect the semantics of functions not only makes manipulation swifter, but permits rapid, natural access to previously developed information. This retrieval and decomposition of expressions is accomplished by template pattern matching (Teitelman 1967).

2. Decomposition

Decomposition occurs during the process of assigning arguments to functions or interpreting assignment statements. In the assignment statement, the expression on the left must match (be an instance of) the expression on the right. For example, the interpretation of the statement

```
(SETQ (SET ←X ←Y .. )(SET RED BLUE GREEN YELLOW))
```

assigns one of the four color words to both X and to Y. The double dots denote a fragment, and permit the sets to be of different

cardinality. Since sets are involved, X and Y may be assigned the same word or different words.

3. Retrieval

The statement

(EXISTS (SET ←X RED ...))

will retrieve from the data base all sets that contain the word RED; X is then assigned some element from one of the retrieved sets. This form of pattern matching permits programs to be nondeterministic. The program may signal that an incorrect choice was made by executing a FAIL. The interpreter is then required to make an alternative assignment. The backtracking necessary to interpret the programs is handled automatically by the interpreter.

More important than backtracking, however, is the fact that all queries into the data base are in the form of associative addressing. Moreover, the search may go two ways. That is, given (P ←K) a program may find all expressions such as (P A) or (P B) or (P ←Z) that match (P ←X). Given (P A), on the other hand, it may find (P ←X) or (P ←Z). This second kind of search permits programs to retrieve axioms about concrete statements.

III CONTROL

Next we would like to discuss some problems of program control. By control we mean local decisions and tactics such as variable bindings, the scope and retention of the bindings, the use of multiple process, and the notion of indecision in program formulation.

A. Context

1. Motivation

To establish an implication, a natural method is to assume the antecedent and attempt to establish the consequent. The task may be to prove a logical implication such as:

If X is a prime, then X is odd.

Or the task may be to establish a causal implication, such as:

If the A-register is incremented by 1, then its value is changed.

In either case a problem-solving strategy must create a "context," make hypothetical assumptions, and derive conclusions (Kalish and Montague 1964). When it is finished, however, it must erase intermediate results, for they depend on the truth of the antecedent, which may not be true even though the implication has been established.

2. Example

This ability of programs to manipulate contexts independently of their own dynamic variable bindings is illustrated by the following program sketch:

TO PROVE $X \rightarrow Y$ where X and Y are any expressions
with respect to context C

ESTABLISH A NEW CONTEXT C′

ASSERT X WRT C′

GOAL, PROVE Y WRT C′

ERASE C′

ASSERT $X \rightarrow Y$ WRT C .

Here X is TRUE only in context C′. All changes in the global data base, including any side effects made during the proof of Y, are erased when C′ is erased. Thus if the user wishes, he may manipulate properties of expressions with a binding mechanism that operates without regard to the bindings of his program variables.

3. QA4 Contexts

As strategies such as these are invoked in a QA4 program, they are assigned a "context" in which they operate. All the properties associated with an expression are stored and retrieved with respect to some context. These running strategies may operate independently in parallel, or may cooperate in a high degree of synchronization. The backtracking, side effects, and communication paths of these strategies are highly controllable. Moreover, the control may be handled either automatically by the interpreter, or manipulated by the strategies themselves. Thus the combination of canonical expressions and a context mechanism permits the programmer new freedoms in strategy communication and data retention.

B. Processes

1. Motivation

The effective use of process structures is another important aspect of QA4 programming techniques. These pseudo-parallel processing structures simplify the programming task, and that is the object of our language. Two instances of the use of processes might illustrate their intended use.

2. OR Example

Given the theorem-proving problem

PROVE A OR B

one could begin to prove both A and B in parallel and terminate as soon as one proof finished. Even if both proofs are not physically running at the same time the decomposition of the problem into conceptually parallel processes has simplified the programming task. For the two processes to be effective, however, they must work together. Suppose we use the following strategy:

- Find the best part to work on, say B.
- Start proving it.
- If it progresses rapidly, keep working.
- If not (we may have made a mistake), save the state of this theorem-proving process, and start out on A.

- If A begins to look harder than B, go back to B. But now incorporate the information you have learned from working on A.

We feel that concrete realizations of strategies of this type are necessary for the construction of problem solvers, and that, by using QA4, the strategies can be easily developed.

3. Exists Example

Another especially relevant problem is to prove

$$(\exists X)(P(X) \& Q(X)) \quad ,$$

for some expressions P and Q. In this case, we first find an X that satisfies P. Then we see if it satisfies Q. If not, maybe we should search for an X for Q, and then see if it satisfies P. Each time we redirect our attention, we want to save the state of the current process, and begin where we left off.

4. Summary

What we seek for QA4 programs is not any magical speedup in execution time, but a useful conceptualization of parallel processes for the programmer. We want to encourage the writing of programs that try this, then try that, then try this again, and at each step use both their old information and newly gained information as best they can. But the advantage comes from subdividing the problem, so the programmer is only concerned with a small problem at a time.

C. Indecision

1. Valueless Variables

Many times, in even a simple problem, procrastination is a good heuristic. For example, in the command

move a block down the hall ,

choosing a block before you plan your path may be a poor approach. First, one should plan a route, then look for an appropriate block--maybe one close to the route. However, during the planning, one should keep in mind that eventually a block will be involved. To ease the task of writing programs that must operate this way, QA4 has unbound but usable variables.

Suppose a program has X as a variable. X can be assigned properties--in our case, it might be restricted to being a block. Since X is a QA4 expression, it can have many properties besides its value. X can also be used as an argument to a subroutine. Even if X appears in an expression, say

(AT ROBOT ←X)

X need not have a value. The expression will be bound to the actual argument of the subroutine. The subroutine can examine X and discover that it does not have a value. It may also examine the properties of X and plan accordingly. It may even pass X on to other subroutines or attach more properties to it.

2. Backtracking

Automatic backtracking provides another mechanism for delaying decisions. When a strategy determines that a variable should take its value from a set, but is not certain which element, it can merely make one of the possible assignments and go on. If a later strategy discovers that the choice was incorrect it may FAIL, and the interpreter will backtrack automatically. While this mechanism can be used as a complete depth-first search mechanism, that is not its intended use. The choice of elements should not be arbitrary. There should be a good first decision, with the hope that the system will not backtrack. If it does, the second should surely work.

D. Iteration

Backtracking also plays an important role in iteration. In our problems, iteration is not naturally expressed as a subscript range. Sometimes it is inconvenient to express it as a logical condition. A more natural way to express the iteration might be to say:

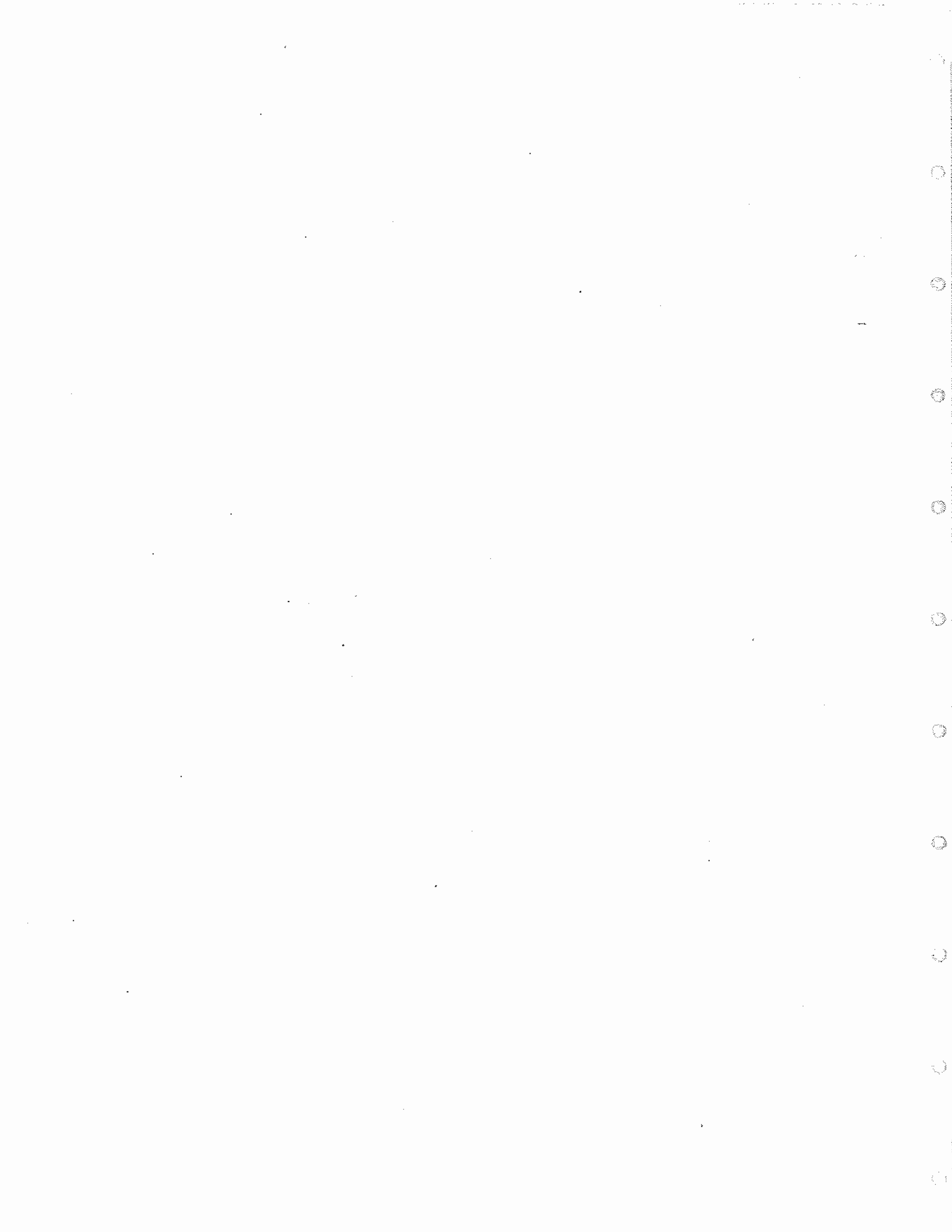
> "Do something for all Xs such that X is the first argument to a certain predicate."

or

> "Do it for all Xs such that X is in this set and X satisfies a predicate."

The REPEAT statement of QA4 provides such a mechanism. With it the programmer can specify the executions of the body of the statement for

all possible ways of doing a particular pattern match, or for all possible expressions in the data base that match a pattern. During each iteration cycle he may also specify which side effects are to accumulate and which are to be removed. Thus the programmer does not have to construct irrelevant data structures. As we have seen, everything in QA4 is geared toward natural, concise expression transformation, even the iteration statements.



IV ORGANIZATION

A. Review of Goals

Remember, the purpose of the QA4 language is to provide a method whereby one can construct programs without having to understand the whole problem or even to have worked out a global structure to the solution process. We expect the programs to grow interactively and to be continually refined and improved. We feel that the programmer has a notion of how the program is to work, but does not understand enough of the notion to write algorithms. If he must express his ideas in standard formal languages the strict formality inhibits his intuition and the ideas are lost. By using QA4, he can express these ideas, ambiguous though they may be. He can write small, individual strategy programs. He may even try out some of the ideas, relying on the interpreter to handle all the ambiguity and make many irrelevant decisions automatically. Then, as he works with the system, the problem solver grows until it handles many cases and appears to have some generality.

Let us now look at some common problem-analysis techniques and how they are expressed in QA4.

B. Goals

1. Motivation

One of the most important problem-solving techniques is the method of using subgoals. The strategy goes like this:

Given a certain goal to satisfy, see if you know the answer. If so, retrieve it and quit.

If not, try to break the problem down into subgoals, and try each one separately.

To encourage this kind of program organization, QA4 provides GOAL statements. To use them, we first write programs that accomplish specific subgoals. The subgoals may be divided into classes. In our automatic program synthesizer, for example, we will have both PROVE goals and SIMPLIFY goals. When our strategies discover new goals, they will say

```
GOAL $PROVE, some exp;
```

or

```
GOAL $SIMPLIFY, some exp;
```

2. <u>How They Work</u>

We first write programs to work on special cases. For example, we write a program that can prove implications by using the conditional derivation method discussed earlier. We identify the structure of the goal the program works on in the pattern that makes up the bound variable of the strategy. Thus our strategy program starts out:

```
(LAMBDA (IMPLIES ←X ←Y) ...)
```

The program also has a name, say CONDER. Now to inform the interpreter that CONDER will solve goals, we add CONDER to the tuple of PROVE programs:

```
(SETQ ←PROVE (CONS $CONDER $PROVE))   .
```

The interpreter now knows that if it is presented with a goal of class PROVE, and if the goal matches the bound variable of CONDER, that CONDER can be used to solve that goal. Later, when we write a program to PROVE conjuncts, it may look like:

```
(LAMBDA (ADD ←A ←B) ...)
```

and be named CONJ. When we state

```
(SETQ ←PROVE (CONS ($CONJ $PROVE))   ;
```

this program also becomes available for working on goals of class PROVE. Since its pattern is different from CONDER, however, it will work on different goals.

3. No Names

During this time, we may have written many programs that have goal statements, and there may or may not be programs available to solve the goals. The main point is that the program may be tried out and tested. If more than one goal-solution program is available, the interpreter will try them in turn and backtrack properly if they fail. The goal programs are not organized in the fashion of standard programming languages. The technique of invoking subroutines is the key. The subroutines are not referenced by their name. Instead, they are called because they accept arguments with a certain structure, and because the programmer claimed that they will solve goals of a certain class.

C. Choosing Solution Programs

It is not enough, however, to use only this single organization technique. Many solution programs may apply, and they must be ordered and selected. Suppose, for example, that the protocol of the problem is to read as though means-ends analysis had been used. Instead of using an executive to perform the analysis, we wish to make every decision on a local level, using pragmatic information. There are many ways of doing this in QA4 programs.

1. Header Tests

The most obvious trick is to put tests at the front of each GOAL program so that it attempts to eliminate itself as soon as possible. For example:

```
(LAMBDA (IMPLIES ←X ←Y)(IF (EQUAL $X FALSE) THEN FAIL) ...)
```

would make this strategy program fail when it is given conditionals with false antecedents. But we would like to avoid initializing and running the program in the first place.

2. Advice

In another method of giving advice to the system, we may specify that a strategy program is to have control over the order and execution of possible goal-solution programs. When these options are used, the strategy program is given the set of choices along with other necessary information. In a manner similar to co-routine execution, this strategy program works with the QA4 interpreter to order and

control further executions. The strategy program may even try the solutions in parallel. This permits one to work for a while, examine the data base, shift ones attention to another, and later resume the former.

D. Models and WHENs

1. Operative Models

We have saved for last a most serious problem: How should a robot planner or theorem prover model the environment it attempts to deal with? To begin, let us consider the two general types of models discussed by Piaget--figurative and operative (Piaget 1960). Figurative models are those in which the objects under consideration--say the blocks in the room--are described by a set of logical statements and general inference rules. That is, what we know about the objects is simply the logical facts immediately at hand and those we could derive with a theorem prover. Operative models, on the other hand, are those in which the objects are modeled through the use of programs. In the case of our blocks, we might have a program for each block. This program could accept messages and give responses. The object is then modeled by its reactions to input. Within this framework, the structure of the object can be directly reflected by the structure of the program. We do not have to go through the intermediate and most-often irrelevant semantics of logic or artificial data structures.

We may even have a set of programs that model a block: one for when it is pushed, one for when another block is placed on it, and yet another for when it is viewed. These programs will no doubt use data structures, and in that sense have figurative data. But the emphasis is now on the program and its current interpretation of the data. The PUSH program may answer many questions, such as:

"Where are you?"

or

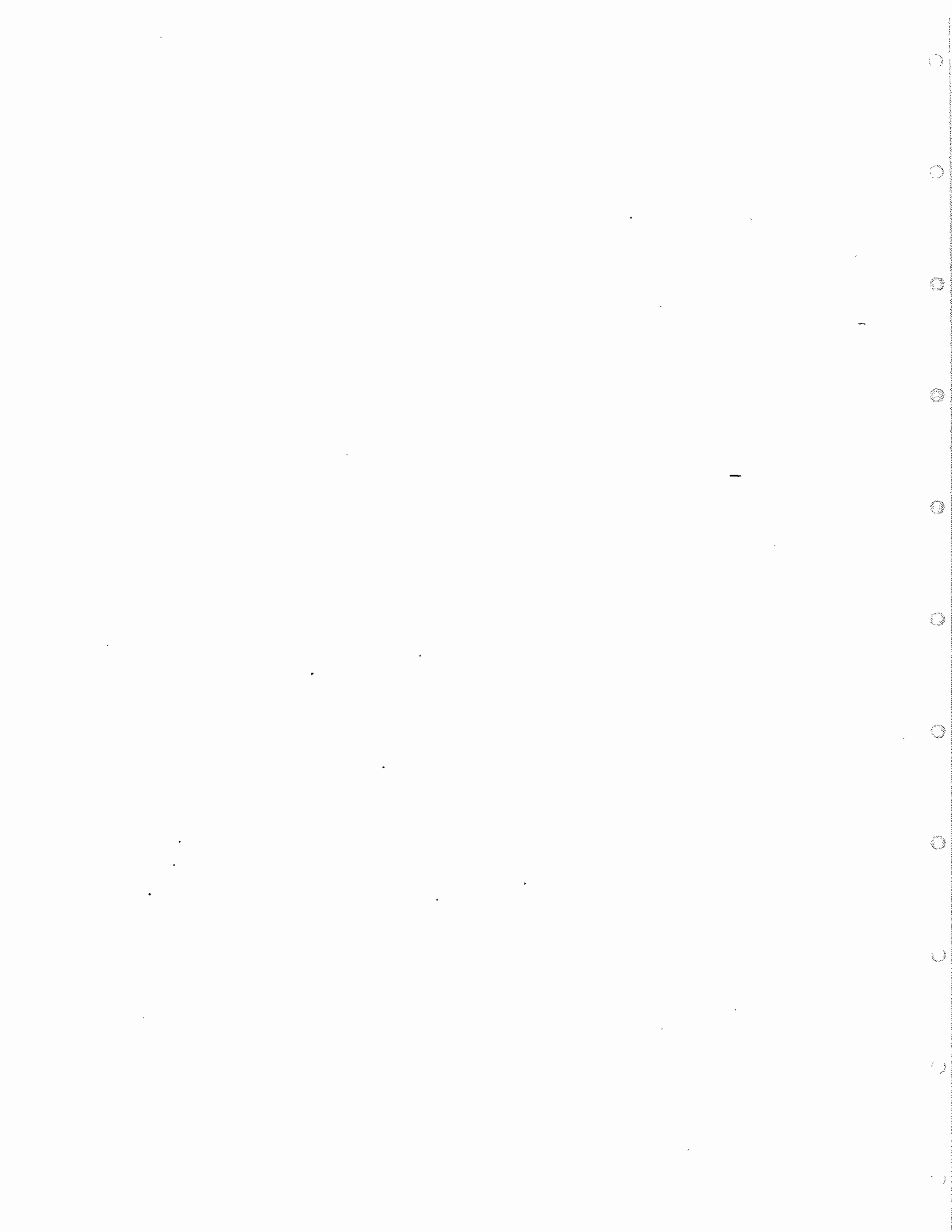
"How long have you been there?" .

It may also answer questions like "Where will you be if I shove you this way?". The information may be readily available in the program-model data base or it may require computation. Thus the position might, at one time, be kept in the coordinates of a room, and at another time, with respect to another block. The answer to the shove question may even be "I will fall over," and the derivation of this answer may exceed our capacity to model blocks in logical statements. The model is now one of action and reaction--using the full power of the QA4 language.

2. WHEN Statements

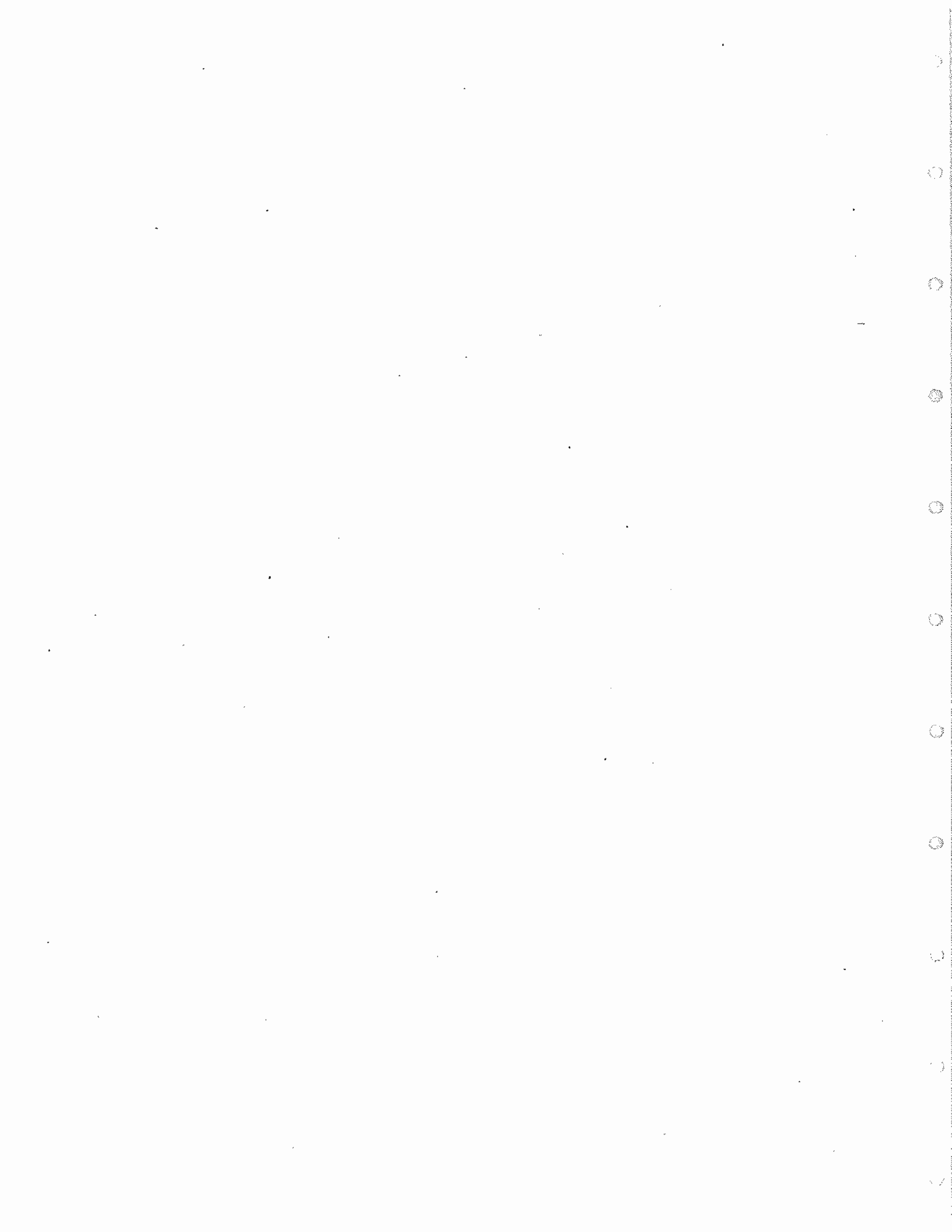
The GOAL mechanism is a help in implementing a problem solver that uses operative models. But the WHEN statement is the basis of the solution. This QA4 command permits us to create a "demon." The demon is assigned a set of watching posts. For example, it is assigned

to watch all expressions that match a certain pattern. When information goes past its post that matches a second pattern, and satisfies that pattern's predicate, it may take control and execute programs. In our example, we may have programs that watch operations on boxes. When things are done to the boxes, these programs modify local data, or invoke yet other demons. In this way, the main method of keeping the model up to date while taking into account complex interactions between the objects is through the use of WHEN programs.



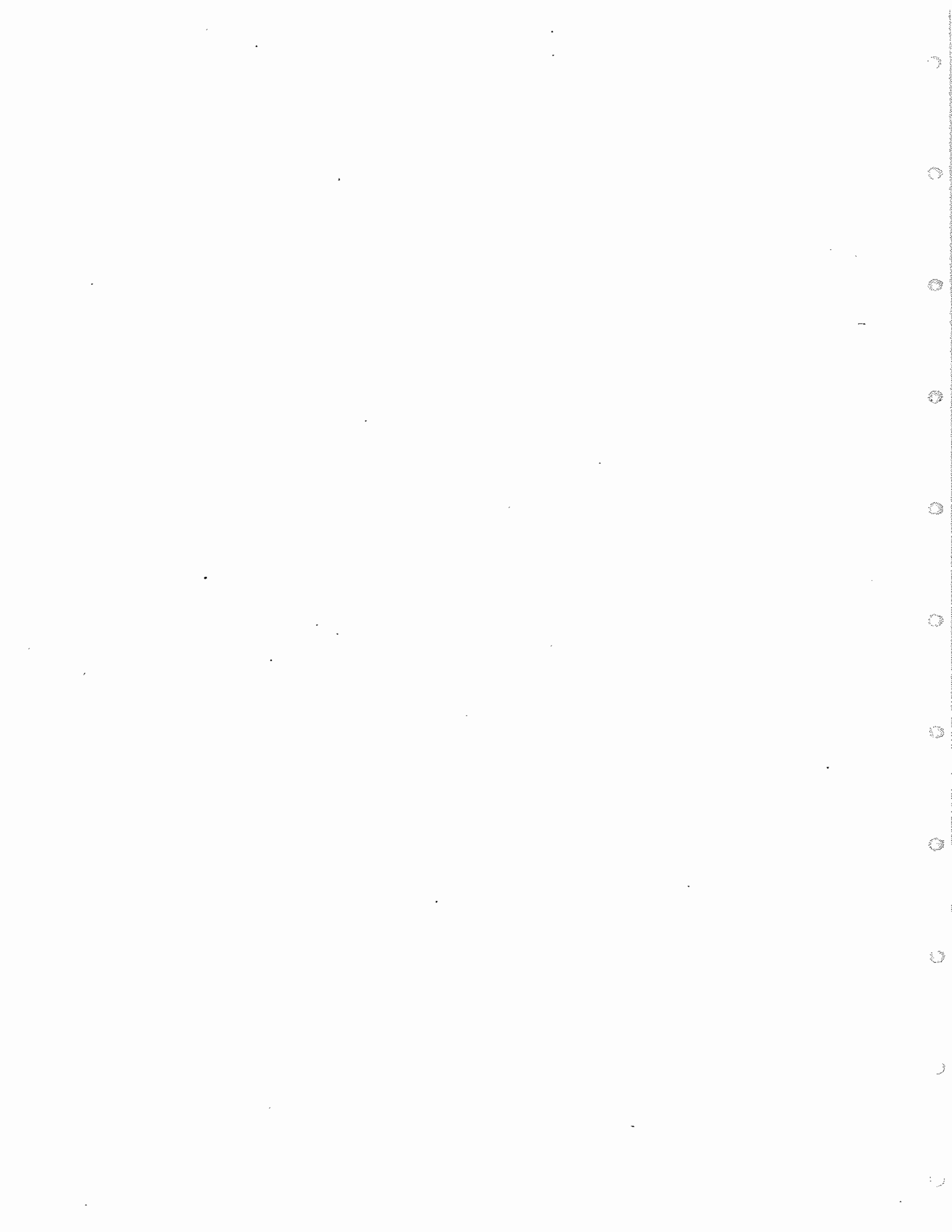
V CURRENT STATUS

An interpreter for the language described herein appears to work properly. This first implementation of the language is extraordinarily general. Every possible step has been taken to be sure that, if we wish, statement operation can be easily modified. This results in slow execution time, but we feel it is worth the price. We will not know just how GOALSs, WHENs, and control structures should operate until we have successfully written some major problem solvers. So we look at the project as a design process that should converge. We have made a first pass at the language. This has permitted us to construct some small problem solvers and discover more precisely what the language should be. As we learn more, we will change the language. And hopefully, we will uncover aspects of designing and building problem solvers at the same time that we discover more about theorem proving, program synthesis, and robot planning.



Chapter Two

A PRIMER



I EXPRESSIONS

A. Introduction

One of the design criteria of QA4 is that problems in the area of theorem proving, automatic program verification and synthesis, and robot planning are to have a compact and natural formulation. Thus, although the form of QA4 is modeled after that of LISP, we find a greater variety of syntactic types in QA4 than in most programming languages: e.g., there are sets, ordered and unordered tuples, lambda expressions, quantified expressions, and function applications.

The basic building blocks of the QA4 language are the primitive expressions. Each utterance in the language is made up of these expressions which we introduce in the next section. Each expression has a "value," which is also a QA4 expression. A basic evaluation function associates with each expression either a number, identifier, or other QA4 expression.

B. The Primitive Expressions

(Identifiers, numbers, truth values, tuples, sets, bags, applications, variables, bound variable expressions, special forms.)

1. Identifiers

An identifier is a string of letters and digits such as X, MARLENE, and A53QQ. The identifiers are used as constants, function

names, predicate names, variable names, and labels of the language. The value of a constant is the constant itself.

2. Numbers

Conventions of the host LISP systems are used, e.g., 3, 0, -2.7 are numbers.

3. Truth Values

There are two special identifiers, TRUE and FALSE, denoting true and false respectively.

4. Tuples

A tuple expression is an expression of the form (TUPLE el ... en). Tuples are the QA4 version of LISP's lists, e.g., (TUPLE A 1 2 3), (TUPLE A JIM (TUPLE 4 D)). The value of a tuple is the tuple whose elements are the values of the elements of the original tuple.

5. Sets

A set expression is an expression of the form (SET el ... en). Since both the order of the elements and the number of occurrences of an element in a set are immaterial, the sets

(SET A B C) (SET C A B) and (SET C A A B C)

are treated as identical expressions. Note that the set (SET C A A B C) never could occur internally as a QA4 expression because each time a set is created, the system reduces and reorders the set, putting it into a normal form. The value of a set is a set made up of the evaluated elements of the original set.

6. Bags

A bag is an expression of the form (BAG el ... en). A bag may be considered to be an unordered tuple or to be a set with multiple elements, e.g., (BAG 1 1 3 5), (BAG A (TUPLE 2 3) 7 KATHY 7). Note that (BAG 2 3 2) = (BAG 2 2 3) ≠ (BAG 2 3). Bags are used as arguments of functions that are commutative and associative, such as PLUS and TIMES. The value of a bag is the bag whose elements are the evaluated elements of the original bag.

7. Applications

An application is an expression of the form (f a). The value of an application (f a) is the result of applying the value of f, which is expected to be a function, to the value of the argument a. Internally, all QA4 functions take one argument; if n arguments are desired they must be grouped into the tuple (TUPLE arg1 arg2 ... argn). Functions can also have sets, bags, or any other type of expressions as argument. An abbreviation convention exists so that if F takes a tuple as argument, the user can write (F arg1 arg2) instead of (F (TUPLE arg1 arg2)). Furthermore, for certain built-in functions such as EQUAL or TIMES that may take sets or bags as arguments, (F arg1 ... argn) will be taken to mean (F (SET arg1 ... argn)) or (F (BAG arg1 ... argn)), whichever is appropriate. The system supplies, in this case, the additional information. For instance, (PLUS 2 1 3) will be taken to mean (PLUS (BAG 2 1 3)).

8. Variables

A variable is an identifier with prefix ←, ?, $, ⟵, ??, or $$. For instance, $Y and ⟵LINDA are variables. In QA4, variables play more roles than in LISP. The use of pattern matching for decomposition of expressions demands a precise "declaration" of the role of the variable in an expression. We do not want to explain the use of all the prefixes at this point in the text. However, a complete description is given in the sections on pattern matching. Roughly speaking, when we want to use the previously assigned value of a variable we give it the $ prefix, while when we want to assign the variable a new value we use the ← prefix.

9. Bound Variable Expressions

A bound variable expression is an expression of the form (keyword bv-part body), where keyword is LAMBDA, FA ("for all") or EX ("exists"), bv-part or bound variable part is an expression to be used as a pattern, and body is any QA4 expression

Example: (LAMBDA ←X (PLUS $X 2)) .

Thus, bound variable expressions are either lambda expressions or quantified expressions.

The QA4 variable binding mechanism is an elegant extension of LISP lambda binding. The purpose of the bound variable part of a lambda expression is to assign values to one or more variables during the evaluation or analysis of the body of the bound variable expression.

Unlike LISP's variable list, a QA4 bound variable is a general pattern which is matched against the argument of the lambda expression. The pattern contains variables that are bound if the pattern successfully matches the argument. Thus, lambda expressions are functions that can take any QA4 expression as argument.

Example: To evaluate the application

```
((LAMBDA (BAG 2 ←X 3) (TIMES $X 5)) (BAG 3 2 4))   ,
```

first the bound variable part (BAG 2 ←X 3) is matched against the argument (BAG 3 2 4), binding the variable X to 4. Next the body (TIMES $X 5) is evaluated with respect to the new binding. The value of the application is then 20.

Example:

```
((LAMBDA (TUPLE 2 (TUPLE ←X A) ←Y) (TUPLE 2 $X $Y)) (TUPLE 2 (TUPLE 3 A) 4))
```

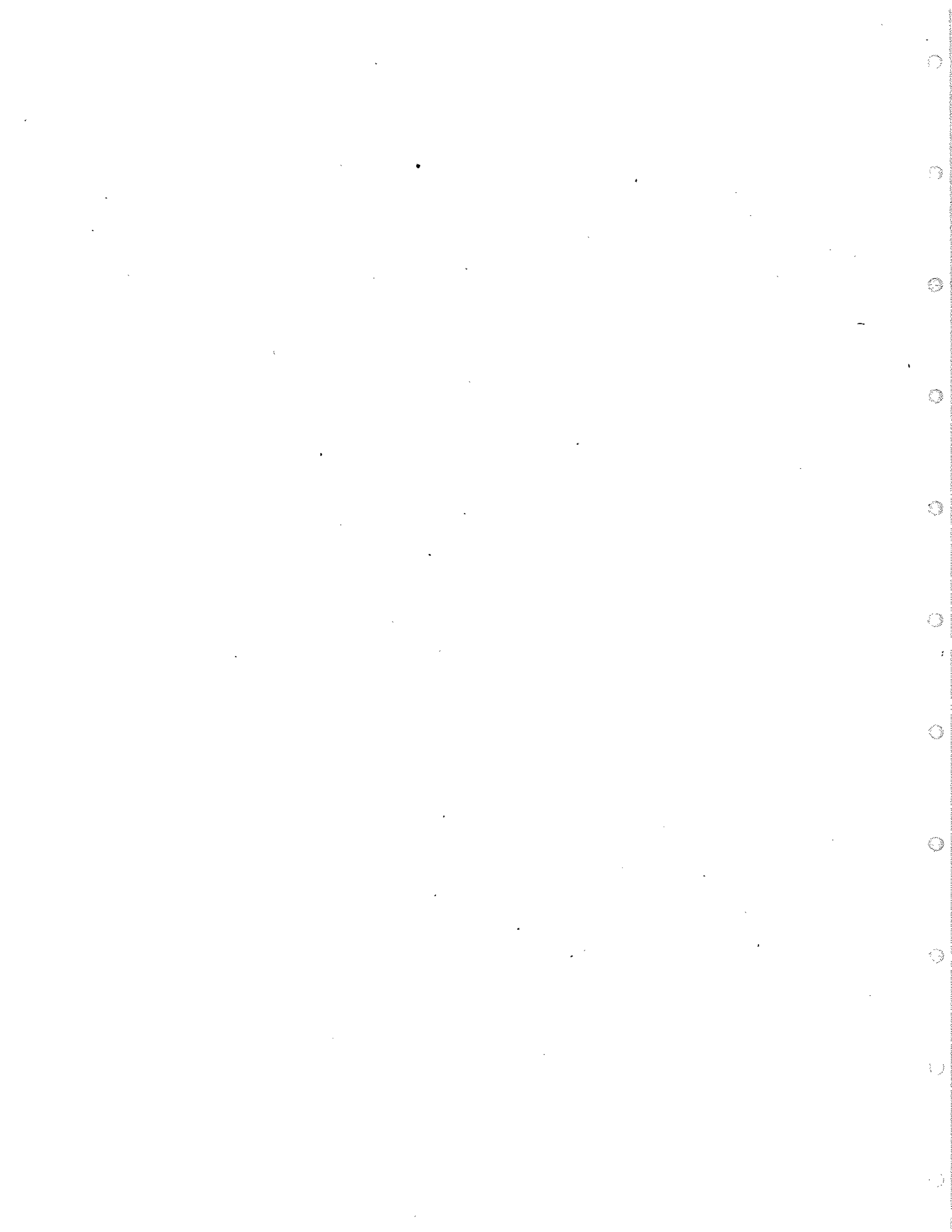
evaluates to

```
(TUPLE 2 3 4)   .
```

For sets and bags, there can be many ways of matching the bound variable pattern against the argument. This possibly leads to some complications which we will discuss in Section III-D.

10. Special Forms

A special form is an expression with a nonstandard method of evaluation. These special forms are also called QA4 statements. For example, the IF statement, the PROG statement, and the GOAL statement are special forms.



II BUILT-IN FUNCTIONS

A. Logical Connectives

Logical operators consider an expression to be TRUE if it has any value other than FALSE. AND, OR, EQUAL, and NOTEQUAL take sets as arguments. In this way the commutativity and idempotency (A & A ⇔ A) of logic operations are built in and need not be stated explicitly. The possibility of expressing logical functions and relations of more than two arguments eliminates many occasions for use of the associativity rule. We do not have to build expressions such as A & (B & C) or (A ∨ B) ∨ C and then express the equivalence of A & (B & C) and (A & B) & C with a rule, but need only write expressions such as

(AND A B C)

(OR A B C)

or (EQUAL E1 E2 E3) .

Recall that we need not include the set indicator for arguments of built-in functions.

- AND

  The connective AND takes a set of expressions as arguments. If none of the elements of this set has the value FALSE, its value is TRUE. (We will express "has value" with the meta symbol ⇒.)

(AND 1 2 3) ⇒ TRUE

(AND $X $Y) ⇒ TRUE when X and Y both are bound to TRUE

(AND A FALSE B) ⇒ FALSE .

- <u>OR</u>

  The connective OR takes a set of expressions as argument and returns ("evaluates to") FALSE if each of the elements of the set has value FALSE; otherwise TRUE is returned.

  (OR TRUE FALSE TRUE) ⇒ TRUE .

- <u>EQUAL</u>

  The connective EQUAL takes a set as its argument and returns TRUE if the values of all of the elements are identical; otherwise FALSE is returned. Note that QA4 functions differ significantly from their LISP counterparts.

  (EQUAL 2 2 2) evaluates to TRUE

  (EQUAL A $X $Y) evaluates to TRUE if X and Y have value A.

- <u>NOTEQUAL</u>

  NOTEQUAL takes a set as argument.

  (NOTEQUAL E1 E2 E3) means E1 $\neq$ E2 & E2 $\neq$ E3 & E1 $\neq$ E3 .

- <u>IMPLIES</u>

  The connective IMPLIES takes a tuple as argument.

  (IMPLIES $A $B $C) means $A $\supset$ $B & $B $\supset$ $C

  where $X $\supset$ $Y is TRUE unless Y has a value FALSE and X has a value other than FALSE.

Examples: (IMPLIES FALSE TRUE) $\Rightarrow$ TRUE

(IMPLIES 2 3) $\Rightarrow$ TRUE

(IMPLIES TRUE TRUE FALSE) $\Rightarrow$ FALSE .

- NOT

NOT takes a single argument and returns TRUE if the value of the argument is FALSE and FALSE otherwise.

(NOT TRUE) $\Rightarrow$ FALSE

(NOT 5) $\Rightarrow$ FALSE

(NOT FALSE) $\Rightarrow$ TRUE .

- IFF

IFF takes a set as argument.

(IFF E1 E2 E3 E4) means E1 $\Leftrightarrow$ E2 & E2 $\Leftrightarrow$ E3 & E3 $\Leftrightarrow$ E4

("$\Leftrightarrow$" stands for "equivalent with.")

IFF asserts that all the members of the set imply each other. An IFF expression is TRUE if none of the set members are FALSE, or if all of them are.

(IFF 2 3) $\Rightarrow$ TRUE

(IFF FALSE FALSE FALSE) $\Rightarrow$ TRUE .

B. Arithmetic Functions

- PLUS

PLUS takes a bag as argument and returns as value the sum of the values of elements of the argument.

Example: (PLUS 1 3 5) $\Rightarrow$ 9 .

This choice of data structure for the argument makes it possible to handle commutativity of addition in a natural way.

Example: (PLUS $X $Y $X 5) and (PLUS $Y $X 5 $X) both have the same value because (BAG $X $Y $X 5) and (BAG $Y $X 5 $X) are indistinguishable in QA4.

- TIMES

  TIMES takes a bag as argument.

  (TIMES 1 5 3 7) forms the product $1 \times 5 \times 3 \times 7$.

  The use of a bag as argument for this operator gives us the same advantages for TIMES as for PLUS.

- SUBTRACT

  SUBTRACT takes a tuple as argument.

  Example: (SUBTRACT 8 3 1 2) means 8 - 3 - 1 - 2.

  (SUBTRACT n1 n2 ... nm) has as value $n1 - (n1 + \ldots + nm)$.

- MINUS

  MINUS forms the arithmetic negative of the value of its argument. (MINUS 4) evaluates to -4.

- DIVIDE

  DIVIDE takes a tuple as argument. (DIVIDE n1 n2 ... nm) evaluates the quotient $n1/(n2 \times n3 \ldots \times nm)$.

  Example: (DIVIDE 16 2 4) $\Rightarrow$ 2 .

- GT

  GT stands for greater than. GT takes a tuple as argument. (GT n1 n2 n3 ...) is TRUE if $n1 > n2$ & $n2 > n3$ ... and FALSE otherwise.

  Example: (GT 10 7 4) $\Rightarrow$ TRUE .

- LT

  LT means less than. It takes a tuple as argument. (LT n1 n2 n3) evaluates to TRUE if $n1 < n2$ & $n2 < n3$ ... and FALSE otherwise.

  Example: (LT 20 20) $\Rightarrow$ FALSE .

- GTQ

  GTQ stands for greater than or equal. GTQ takes a tuple as argument. (GTQ n1 n2 n3 ...) evaluates to TRUE if $n1 \geq n2$ & $n2 \geq n3$ ... and FALSE otherwise.

  Example: (GTQ 5 4 4 2) evaluates to TRUE .

- LTQ

  LTQ stands for less than or equal. LTQ takes a tuple as argument. (LTQ n1 n2 n3 ...) evaluates to TRUE if $n1 \leq n2$ & $n2 \leq n3$ ... and FALSE otherwise.

  Example: (LTQ 24 45 45) evaluates to TRUE .

C. Structural Functions

- IN

IN takes a 2-tuple as argument. (IN var s) evaluates to TRUE if the value of the variable var is an element of the set, bag, or tuple denoted by s.

Examples:

(IN 4 (BAG 5 A 4)) ⇒ TRUE

(IN $Z (SET 1 9 XX YY)) ⇒ TRUE when Z is bound to XX

(IN 9 (TUPLE 6 7 8 9)) ⇒ TRUE .

- CONS

CONS takes a 2-tuple as argument. (CONS x s) returns the results of inserting the value of x as an element of the value of s. If s has a tuple as value, the value of x is inserted at the front. We cannot say anything about the position of the added element if the value of s is a bag or a set.

Example: (CONS JANE (TUPLE JANE CABBAGE 3)) ⇒
(TUPLE JANE JANE CABBAGE 3)

Example: (CONS JANE (BAG JANE 3 SPINACH)) ⇒
(BAG 3 JANE JANE SPINACH))

Example: (CONS JANE (SET JANE CAULIFLOWER 2)) ⇒
(SET 2 JANE CAULIFLOWER) .

- <u>NTH</u>

  NTH takes a 2-tuple, (<u>t</u> <u>n</u>), as argument; the first element, <u>t</u>, is a tuple and the second element, <u>n</u>, is a number. Its value is the $\underline{n}^{th}$ element of tuple <u>t</u>.

  Example: (NTH (TUPLE C B A) 2) ⇒ B .

- <u>APPEND</u>

  APPEND takes a tuple as argument. If the elements are tuples themselves, APPEND concatenates them.

  Example: (APPEND (TUPLE 1 A) (TUPLE A 2) (TUPLE 3 B)) ⇒

  (TUPLE 1 A A 2 3 B) .

  If the elements are sets, APPEND takes their set union.

  Example: (APPEND (SET 1 A) (SET A 2) (SET 2 B)) ⇒

  (SET 1 2 A B) .

  If the elements are bags, APPEND takes their bag union, i.e., it preserves the multiplicities of the elements.

  Example: (APPEND (BAG 1 A) (BAG A 2) (BAG 2 B)) ⇒

  (BAG 1 2 2 A A B) .

- <u>INTERSECTION</u>

  INTERSECTION takes a set of sets as its argument; its value is the intersection of the elements of the argument.

  Example: (INTERSECTION (SET 1 A) (SET A 2)) ⇒ (SET A) .

- <u>DIFFERENCE</u>

DIFFERENCE takes a tuple of sets as its argument. (DIFFERENCE S1 S2 S3 ...) computes the set-theoretic difference between S1 and (APPEND S2 S3 ...); that is, the set of elements which belong to S1 but which do not belong to (APPEND S2 S3 ...).

Example: (DIFFERENCE (SET 1 A 2 B) (SET 1) (SET 2 3 C)) ⇒ (SET A B) .

III PATTERN MATCHING

A. Introduction

Expressions are taken apart and their components are named through the use of the pattern language. Pattern matching takes place during the execution of statements, lambda binding, data base retrievals, and various other operations to be explained in the following sections. Patterns and arguments can be arbitrary QA4 expressions. A simple example is the following assignment:

(SETQ (TUPLE ←X ←Y) (TUPLE 1 2)) .

The pattern (TUPLE ←X ←Y) is matched with (TUPLE 1 2) and variables X and Y will be bound to 1 and 2 respectively. If the pattern matcher cannot match a pattern with an argument, a condition known as failure will occur. This possibility is discussed in the section on failure, VI-D. The same decomposition process as described for SETQ takes place during lambda binding.

Example: (LAMBDA (TUPLE ←X ←Y) (TUPLE $Y $X)) (TUPLE 1 2) ⇒ (TUPLE 2 1).

Example: ((LAMBDA (SET 3 (TUPLE ←X 4)) $X) (SET (TUPLE 5 4) 3)) evaluates to 5.

Example: EXISTS is a built-in QA4 statement to retrieve expressions from the data base. Note that the existential quantifier is called EX and is not the same as the EXISTS construct.

(EXISTS (RED N ←OBJECT)) searches the data base for an object asserted to be red and binds OBJECT to that object.

As an example of the gain of clarity consider the following example: Suppose L is a list of the form (X Y (V W)) and we want to set variable K to the rearranged list (V W (X Y)). In LISP we would write

```
(SETQ K
      (CONS (CAADDR L)
      (CONS (CADADDR L)
            (CONS (CONS (CAR L) (CONS (CADR L) NIL)) NIL))))   ,
```

or, more efficiently,

```
(SETQ M (CADDR L))
(SETQ K
      (APPEND M
              (LIST (LIST (CAR L) (CADR L)))))   .
```

Although the operation described here is conceptually simple, it is quite impossible to see what is going on. In QA4 we would write:

```
(SETQ (TUPLE ←X ←Y (TUPLE ←V ←W))
      $L)
(SETQ ←K
      (TUPLE $V $W (TUPLE $X $Y)))   .
```

The QA4 representation is clearer because it is more pictorial. Notice that the QA4 representation is almost identical to the English wording.

The pattern language makes it possible to specify an expression by giving a partial description. For example, to search for an expression asserting that some door connects two rooms ROOM1 and ROOM2, we execute

(EXISTS (CONNECTS ←DOOR ROOM1 ROOM2)) .

We see here an example of the use of an application (CONNECTS ←DOOR ROOM1 ROOM2) that will not and cannot be evaluated. The expression is always used in quoted form. We will use this technique often in representing predicates in QA4.

When matching sets and bags, an element of nondeterminism can enter.

Example: (SETQ (SET ←X ←Y) (SET 1 2)) .

The two possible assignments are X ⇒ 1, Y ⇒ 2, and X ⇒ 2, Y ⇒ 1.

The pattern matcher of the QA4 language implementation is able to recognize the different alternatives, to choose one, and to produce the "next" possible set of bindings on request. How this nondeterminism is used for programming purposes will be explained in the section on program control, Section VI.

B Patterns

We will now introduce the reader to the variety of patterns that can be used in QA4, and to their interpretation.

1. Constants

Constants are identifiers without prefixes. A constant occurring as a subpattern in a pattern will match only another instance of itself, e.g., (SETQ A A) succeeds, although it does not actually rebind anything. (SETQ A B) fails.

2. Variables

A variable is an identifier prefixed by one of the symbols $, ←, ?, $$, ←←, ??.

Example: ←X

?Y

??COLEEN

←←IDENT .

Note that ←X and $X represent the same variable with different prefixes reflecting their differing roles. Variables are used in a variety of ways in QA4. Variables need prefixes in order to resolve such ambiguities in pattern matching as the following: Suppose the language did not use prefixes, and we executed the statements: (SETQ X 4) and (SETQ (TUPLE X Y) (TUPLE 1 2)). Should the match of (TUPLE X Y) with (TUPLE 1 2) fail because X already has value 4, or should the match succeed, rebinding X to 1 and binding Y to 2? In QA4 we solved this dilemma by explicitly stating with the aid of variable prefixes what action we want to be taken, as we will see in the next section.

Prefixes are also necessary to distinguish variables from constants: Unprefixed identifiers always have themselves as value.

3. The Prefixes

We will now discuss the various prefixes in detail.

a. ←var

The prefix ← permits its variable to take a new value.

Example: (SETQ ←X 4) will bind X to 4.

(SETQ (TUPLE ←X 2) (TUPLE 1 2)) will bind X to 1, overriding the previous binding of X to 4.

The new value of the variable is retained after the successful completion of a match. If the match fails or if a later program failure occurs, the old value of the variable is maintained. Note that within a single expression different occurrences of a variable, X, must all be bound to identical expressions: e.g.,

(SETQ (BAG ←X ←X) (BAG 2 2)) will succeed in binding X to 2 but

(SETQ (BAG ←X ←X) (BAG 2 3)) will fail.

This restriction also holds for variables with other prefixes as well.

b. ?var

Variables prefixed by ? match a single element. ?var matches an expression exp if

- var already has value exp
- var has no value.

In these cases var will be bound to exp after the match has taken place. If var has any value other than exp, the match will fail. Notice that ?var only differs from ←var when var has a value.

Example: (SETQ ←X 5)

(SETQ (TUPLE 3 ?X) (TUPLE 3 5)) succeeds, but

(SETQ ←X 5)

(SETQ (TUPLE 3 ?X) (TUPLE 3 2)) fails.

c. $var

Whereas variables prefixed with ← or ? can only appear in patterns, variables prefixed with $ can appear at arbitrary places in QA4 expressions. Furthermore, a variable with prefix $ is generally expected to have a value when it is used. $var matches exp if and only if var already has value exp.

Example: The sequence (SETQ ←X 4)

(SETQ (TUPLE $X 2) (TUPLE 1 2)) fails,

but if X had been bound to 1 with

(SETQ ←X 1)

the match would have succeeded.

In an expression other than a pattern to be matched, S_var_ is taken to mean the value of _var_. Thus, the sequence

(SETQ ←X B)

(SETQ (TUPLE A B) (TUPLE A $X)) will succeed.

Typically, lambda expressions have variables with prefix ← in the bound variable part; if we want to refer to the bindings of those variables in the body of the lambda expression, we use the same variables with prefix $.

Example: Suppose we want to define a simplification rule for addition. The function defined by (LAMBDA (ADDITION ←X 0) $X) will have output 12 when applied to (ADDITION 12 0).

The value of a variable may be a function. Thus, we can define a function by setting a variable to a lambda expression.

Example: After executing the sequence

(SETQQ ←SIMP (LAMBDA (ADDITION ←X 0) SX))

where SETQQ is a SETQ which does not evaluate its second argument,

(SETQ ←Y ($SIMP (QUOTE (ADDITION 5 0)))) ,

Y will be bound to 5.

d. <u>Summary of Prefix Types</u>

We may summarize the differences between the prefixes in the following table representing the result of matching the variable X

against the constant A. The vertical axis represents the prefix of X, and the horizontal axis represents the QA4 value of X before matching; "unbound" means the variable has no value. The item in the table represents the value of X after the match, where NIL means a failure of the match. If a match fails, the variable always has the value it had previous to the attempted match.

X matched against A, for example (SETQ ←X A)
(SETQ ?X A)
or (SETQ $X A)

	A	unbound	B
←X	A	A	A
?X	A	A	NIL
$X	A	NIL	NIL

←←VAR, ??VAR, $$VAR; Fragment Variables

The rules for matching a variable prefixed by ←←, ??, or $$ are analogous to those for ←, ?, and $ prefixes. However, variables with these prefixes will be bound to a fragment of a set, tuple, or bag, rather than to a single element. These variables are called fragment variables.

Example: (SETQ (TUPLE ←←X 4) (TUPLE 1 2 3 4)) will bind X to the fragment (TUPLE 1 2 3) of (TUPLE 1 2 3 4).

Example: If the statement (SETQ (SET 2 ←←X) (SET 1 2 3 4))

is executed, $X will then evaluate to the fragment (SET 1 3 4) of (SET 1 2 3 4).

C. The Fragment Variable Applied to Construction

A fragment variable can also be used in constructing a tuple, set, or bag. For instance, suppose we are constructing a tuple one of whose elements is a fragment variable, with prefix $$. The system then assumes that the variable is itself bound to a tuple, whose elements are then added to the original tuple.

Example: (LAMBDA (TUPLE ←←X 4) (TUPLE 0 $$X)) applied to

(TUPLE 1 2 3 4) will bind X to (TUPLE 1 2 3).

The body (TUPLE 0 $$X) evaluates to (TUPLE 0 1 2 3). If the body were (TUPLE 0 $X), the value of the application would be (TUPLE 0 (TUPLE 1 2 3)).

Example: (LAMBDA (SET A ←←X) (SET $$X 1)) applied to

(SET A B C D) will evaluate to (SET 1 B C D).

Note that the same variable can be used as a fragment variable after having been bound as an individual variable.

Example: After executing (SETQ ←X (BAG A B))

(SETQ ←Y (BAG B $$X A)), Y will have value

(BAG A A B B).

D. Multiple Matches

By admitting patterns with sets, bags, or fragments, we have allowed the possibility that a pattern may match the same expression in more than one way.

Example: (SETQ (SET ←X ←Y) (SET 1 2)) can either bind X to 1

and Y to 2, or X to 2 and Y to 1.

When such nondeterministic matches are found, the system will choose one of the possible matches. The alternative matches are made available through the failure and backtracking mechanism, which will be explained in a later chapter. Now let us examine some other ways in which nondeterministic matches can occur.

1. Tuples

Nondeterministic matching occurs with the use of more than one fragment variable.

Example: (TUPLE ←←X ←←Y) ≈ (TUPLE 1 2 3 4)

(≈ means "matches.")

with matches X ⇒ (TUPLE) Y ⇒ (TUPLE 1 2 3 4)

X ⇒ (TUPLE 1) Y ⇒ (TUPLE 2 3 4)

X ⇒ (TUPLE 1 2) Y ⇒ (TUPLE 3 4)

X ⇒ (TUPLE 1 2 3) Y ⇒ (TUPLE 4)

and X ⇒ (TUPLE 1 2 3 4) Y ⇒ (TUPLE)

Example: (TUPLE ←←X A ←←Y) ≈ (TUPLE A B C A D E) matches

X ⇒ (TUPLE), Y ⇒ (TUPLE B C A D E)

X ⇒ (TUPLE A B C), Y ⇒ (TUPLE D E)

2. Sets

In sets and bags, only one fragment variable is allowed. Nondeterministic matches occur with and without the use of fragment variables.

Example: (SET ←X ←Y ←Z) $\approx$ (SET 1 2 3) matches

X ⇒ 1, Y ⇒ 2, Z ⇒ 3

X ⇒ 2, Y ⇒ 1, Z ⇒ 3, etc.

Example: (SET ←X ←Y ←Z) $\approx$ (SET 1 2) with matches

X ⇒ 1, Y ⇒ 1, Z ⇒ 2

X ⇒ 1, Y ⇒ 2, Z ⇒ 1

X ⇒ 1, Y ⇒ 2, Z ⇒ 2

X ⇒ 2, Y ⇒ 1, Z ⇒ 1

X ⇒ 2, Y ⇒ 1, Z ⇒ 2

X ⇒ 2, Y ⇒ 2, Z ⇒ 1 .

Note that two or more individual variables can be matched against the same set element. However, if an element is in a set fragment to which a fragment variable has been bound, no other variable may be bound to that element.

Example: (SET ←X ←Y ←←Z) (SET 5 6 7) matches

X ⇒ 5, Y ⇒ 5, Z ⇒ (SET 6 7)

X ⇒ 5, Y ⇒ 6, Z ⇒ (SET 7)

X ⇒ 5, Y ⇒ 7, Z ⇒ (SET 6)

X ⇒ 6, Y ⇒ 5, Z ⇒ (SET 7)

X ⇒ 6, Y ⇒ 6, Z ⇒ (SET 5 7)

X ⇒ 6, Y ⇒ 7, Z ⇒ (SET 5)

X ⇒ 7, Y ⇒ 5, Z ⇒ (SET 6)

X ⇒ 7, Y ⇒ 6, Z ⇒ (SET 5)

X ⇒ 7, Y ⇒ 7, Z ⇒ (SET 5 6) .

However, the binding X ⇒ 5, Y ⇒ 6, Z ⇒ (SET 6 7), for instance, is not allowed.

3. <u>Bags</u>

In bags, as in sets, only one fragment variable is allowed. The conventions for matching bags are slightly different from those for matching sets. In bags we can have multiple occurrences of elements.

Example: (LAMBDA (BAG ←X ←Y ←X) body) (1 2 2) will bind X to 2 and Y to 1 with the body of the lambda expression as scope.

Two variables in a bag cannot match the same element. Thus, (BAG ←X ←Y) does not match (BAG 1), whereas (SET ←X ←Y) does match (SET 1)

Example: (SETQ (BAG ←X ←Y ←Y ←X) (BAG A B A B)) will bind X to A, Y to B, or X to B, and Y to A.

Example: (SETQ (BAG ←X ←X ←←Y) (BAG 1 2 3 1 3 2) matches

X ⇒ 1, Y ⇒ (BAG 2 3 3 2)

X ⇒ 2, Y ⇒ (BAG 1 3 1 3)

X ⇒ 3, Y ⇒ (BAG 1 2 1 2) .

E. Subpatterns

1. Tuples, Sets, and Bags as Elements of a Pattern

Patterns can have patterns as elements. Sets may have tuples as elements, bags may have sets and tuples as elements, etc.

Example: The statement (SETQ (TUPLE ←←X (TUPLE 3 4)) (TUPLE 1 2 (TUPLE 3 4))) will bind X to (TUPLE 1 2).

Example: (LAMBDA (SET 2 (BAG ←X ←X) (BAG ←Y ←Y)) body) applied to (SET (BAG 1 1) 2 (BAG 3 3)) will give the matches X ⇒ 1, Y ⇒ 3 and X ⇒ 3, Y ⇒ 1.

Example: (TUPLE (SET ←X ←Y) (SET ←Y ←Z)) matches (TUPLE (SET A B) (SET C B)) with

X ⇒ A

Y ⇒ B

Z ⇒ C .

2. Finding Subexpressions

A pattern of the form (... pat), where pat is a pattern, matches an expression if pat matches some subexpression of that expression.

Example: (.. 2) ≈ (TUPLE (SET 2) 1)

(.. (BAG 2 3)) ≈ (BAG 2 3)

(TUPLE (.. ←X (.. ←X))) ≈ (TUPLE (SET A (TUPLE B))

(BAG C (TUPLE A))) with X ⇒ A.

Use of the ... construct may result in nondeterministic matches.

Example: (.. (TUPLE ←X (SET ←Y))) ≈ (TUPLE A (SET (TUPLE B

(SET C)))) with X ⇒ A, Y ⇒ (TUPLE B (SET C))

or X ⇒ B, Y ⇒ C.

IV EVALUATION AND INSTANTIATION

In this section we will discuss the way statements and function applications are evaluated, and show methods for altering the mode of evaluation.

When an application of the form (f x) is evaluated, first x and then f is evaluated, and the value of f is applied to the value of x.

Example: (SETQ ←X ($F $Y 2))

If F is bound to PLUS and Y to 5, this statement will bind X to 7.

However, statements do not ordinarily evaluate their arguments but rather instantiate them. To instantiate an expression means to replace the $ and ? variables by their values.

Example: (RETURN ($F $Y 2))

will return the value (PLUS 5 2) if F is bound to PLUS and Y to 5.

An expression can be given a MODELVALUE. This MODELVALUE is entirely independent of the value of an expression which is discussed above. MODELVALUES can be assigned by a user to an expression with a PUT or ASSERT statement (see Section IV-D).

- <u>Quote</u>

The QUOTE statement has the format (QUOTE e). When this statement is evaluated it does not evaluate the expression e but merely returns the unevaluated e as its value. The QA4 quote is the same as the LISP QUOTE.

- <u>The Quasi-Quote ′</u>

The quasi-quote ′ has the format (′e) (Quine, 1965). This statement instantiates e and returns the instantiated expression as its value.

Example: (′ (PLUS $X 5)) ⇒ (PLUS 2 7) when X has value 2.

- <u>The Forced Eval =</u>

If we want to evaluate an argument of a statement, which normally would only be instantiated, we can apply the forced eval operator = to the argument. Then the statement will be applied to the evaluated argument.

Example: (RETURN (= ($F $Y 2))

will return 7 if F is bound to PLUS and Y to 5.

- <u>The Instantiation Inhibitor ":"</u>

If we want to prevent the instantiation of a $ variable that appears in an expression that is to be instantiated, the prefix : is added.

Example: (′ (:$GOTOACTION ROBOT $X)) ⇒ ($GOTOACTION ROBOT ROOM1)

where X is bound to ROOM1.

V THE NET: ASSOCIATING INFORMATION WITH EXPRESSIONS

The QA4 expression may have many components. The character string that we normally call an "expression" is only one of the components, namely, the syntactic form, of a complete QA4 expression. Other components of an expression can be its MODELVALUE, how many times it has been accessed, or other arbitrary information entered by the user.

A. Internal Format of QA4 Expressions

The assertion that a box is in a certain room can be represented by a QA4 expression with syntactic component (INROOM BOX1 ROOM1) and value component TRUE. We may add other components too, for instance, an indication as to how this expression was derived. The whole expression is represented as a property list with the syntactic component of the expression stored under an indicator. A QA4 expression might look like:

```
(NETEXPRESSION NAME 43
               EXPV (SET 1 2 3)
               COLOR RED
               LENGTH 3 ...)    .
```

Properties are of two kinds:

- Syntactic--These are never changed by the user once the expression is created. Examples are:

  OCCURSIN, a list of pointers to the superexpressions;

  NAME, a number uniquely assigned to this expression by the system.

- Semantic--These may be changed by the user at any time. Examples might be: MODELVALUE, the indicator for a user--assigned value of an expression; COLOR, SIZE.

B. Entering a New Expression in the System

When a new syntactic form is typed in or constructed by a program, it is entered into a discrimination net. A property list is constructed having the syntactic form as one of its properties, under the indicator EXPV; this property list is the QA4 expression, and it becomes a new node in the net. If, at some later time another form is entered that is identical to the first, considering possible set and bag permutations and change of bound variables, then the new form is recognized as being the same as the old and no new QA4 expression is constructed.

For instance, suppose we enter a new syntactic form (COLOR FLOWER YELLOW). Then the net is searched for other instances of the same form. Finding none, the system constructs a QA4 expression with syntactic component (COLOR FLOWER YELLOW) and inserts it as a new node in the discrimination net (Minsky 1963).

Suppose that at some later time the same form (COLOR FLOWER YELLOW) is constructed. The system then passes this form through the net and identifies it with the QA4 expression it has already constructed, rather than constructing a new one.

C. Advantages of the Net Storage Mechanism

The net mechanism makes it possible to store properties on expressions in the same way that LISP stores properties in atoms. We may pass a given syntactic form through the net to retrieve the property list on which that form appears. This property list is the QA4 expression whose syntactic component is the given form.

The discrimination net is also valuable for retrieving items from the data base. Given a pattern, we can search the net for a QA4 expression whose syntactic component is matched by the given pattern.

We will now describe the QA4 statements that search the net and manipulate property lists.

D. Property List Operations

A series of operations is available to add or retrieve a property under a certain indicator if the syntactic component of the expression is given.

Example: (ASSERT (P A)) puts the property TRUE under indicator MODELVALUE of the expression whose syntactic component is (P A).

These expressions first retrieve the expression by dropping the given syntactic form into the net. The expression is found on one of the terminal nodes of the net and the appropriate action is taken.

These statements do not evaluate their arguments, but they do instantiate them; i.e., $ variables are replaced by their values, and ? variables that have values are replaced by those values.

1. PUT

Format: (PUT syntactic-form indicator property).

This statement puts the property under the indicator on the expression with the given syntactic-form. The property is returned as the value.

Example: When (PUT (TUPLE A B) FATHER 24) is executed, the expression

(... EXPV (TUPLE A B) ...) becomes

(... EXPV (TUPLE A B) ... FATHER 24 ...),

and 24 is returned as value.

Example: (PUT (CLIMB $Y) USE $TABLE). Suppose $Y has value (BAG 2 2) and $TABLE has value (TUPLE 1 A 2 B 3 C), then the expression

(... EXPV (CLIMB (BAG 2 2)) ...)

will be retrieved and transformed into

```
( ... EXPV (CLIMB (BAG 2 2)) ...
     USE (TUPLE 1 A 2 B 3 C) ...)   .
```

2. GET

Format: (GET syntactic-form indicator)

This statement retrieves the expression, looks up the property under the given indicator and returns this property.

Example: If X has value A, and if the PUT in the first examples of section 1 has been evaluated, (GET (TUPLE $X B) FATHER) returns value 24.

3. <u>ASSERT</u>

Format: (ASSERT <u>syntactic-form</u>).

ASSERT is a PUT statement which puts property TRUE under indicator MODELVALUE of the expression whose syntactic component is the given syntactic form.

Example: (ASSERT (P A)) ⇒ TRUE

Example: (ASSERT (P $X)) ⇒ TRUE, with X ⇒ A.

This assertion is identical to the first one.

Example: If R is bound to ON (ASSERT ($R CUP SAUCER)) puts property TRUE under indicator MODELVALUE of the expression corresponding to (ON CUP SAUCER).

4. <u>DENY</u>

Format: (DENY <u>syntactic-form</u>) puts the property FALSE under the indicator MODELVALUE on the property list of the expression.

Example: (DENY (RAINY WEATHER))

Example: (DENY (RED CUBE))

Note that (GET (RED CUBE) MODELVALUE) will now return FALSE.

5. <u>SETQ</u>

Format: (SETQ <u>pattern</u> <u>exp</u>)

The SETQ statement matches <u>pattern</u> against the result of evaluating <u>exp</u>. The variables in the pattern are bound to the expressions against which they match.

A variable, like other syntactic forms, is the syntactic component of a QA4 expression, which is a property list. When a variable is bound to an expression, that expression is stored as a property under the indicator MODELVALUE.

Example: (SETQ ←X 4)

4 will be stored under indicator MODELVALUE of the QA4 expression whose syntactic component is X.

Example: (SETQ (TUPLE ←X ←Y) (TUPLE 1 2)).

For more examples see Section IV on patterns. Note that if the match is nondeterministic, more than one assignment can be made. However, the system will choose one assignment, while the alternative choices for assignments are available through the backtrack mechanism (Section IV-D-1).

6. <u>SETQQ</u>

Format: (SETQQ <u>syntactic-form</u> <u>exp</u>)

Identical to SETQ except that this statement does not evaluate <u>exp</u>. SETQQ is handy for defining functions.

Example: (SETQQ ←ADD1 (LAMBDA ←X (PLUS $X 1))).

7. EXISTS

Format: (EXISTS pattern ind1 prop1 ind2 prop2 ...)

The EXISTS statement tries to find an expression in the net whose syntactic component is matched by the pattern, and whose property under indicator ind1 is prop1, under indicator ind2 is prop2, and so forth. If no indicators and properties are included as arguments to EXISTS, the system behaves as if indicator MODELVALUE and property TRUE were specified as arguments. If the statement (EXISTS pattern IGNORE MODELVALUE) is executed, the MODELVALUE of the net expressions will be disregarded. In case more than one match is possible, one of the forms is cnosen to be the value. In this case a "backtracking point" is established which makes it possible to obtain the other matching syntactic forms. If no expression is found, a "failure" occurs. When an expression is found, the variables of pattern are bound to the appropriate parts of the expression. (For backtracking and the FAIL mechanism see Section VI on program control.)

Example: (EXISTS (P ←X)) ⇒ (P A) if (P A) has been asserted previously.

Example: (EXISTS (←F 4) COUNT 20) ⇒ (ADD1 4) if (ADD1 4) is in the net with COUNT 20 on its property list.

8. <u>INSTANCES</u>

Format: (INSTANCES <u>pattern</u> <u>ind1</u> <u>prop1</u> <u>ind2</u> <u>prop2</u> ...)

The INSTANCES statement is similar to EXISTS, but it returns a set whose elements are all possible matching syntactic forms. The values of variables are unchanged.

Example: (INSTANCES (P ←X)) ⇒ (SET (P A) (P 4) (P (ADD1 23)))

(INSTANCES (FA ←X (OR (←P $X) (NOT (←P ←X))))) ⇒

(SET (FA Z (OR (HUMAN Z) (NOT (HUMAN Z))))

(FA Y (OR (BIG Y) (NOT (BIG Y))))) .

All statements in this section have an optional last argument. This argument specifies a context. The section on contexts will explain this feature in detail.

VI PROGRAM CONTROL

A. Overview

QA4 offers some unusual ways of governing the flow of control in a program such as backtracking and pattern-directed function calls. These techniques have proven to be useful in so many problem-solving programs that it is worthwhile to include them as language features.

B. Backtracking

1. Setting Up a Backtracking Point

Whenever any indeterminacy occurs during the execution of a program, the system makes an arbitrary choice between the alternatives, and, only on user request for SETQ and lambda expressions, establishes a "backtrack" or "choice" point in the program (Golomb 1965). At a backtrack point a "snapshot" of the world is taken, which makes it possible to restore the state of the system as it was when the point was created, make an alternative choice, and resume execution from immediately after the backtrack point.

Example: Suppose we execute (EXISTS (TYPE ←N BOX)), and that we have previously asserted (TYPE A BOX), (TYPE B BOX), and (TYPE C BOX). The system establishes a choice point at this position in the program. One of the first three assertions is chosen, say (TYPE A BOX), N is bound to A, and execution resumes. If a failure occurs later in the program, control returns

to the choice point, the state of the system is restored to what it was when the point was established, and a different assertion is chosen, say (TYPE B BOX). Then N is bound to B, and execution continues immediately after the EXISTS statement.

Example: (SETQ (SET ←X ←←Y) (SET A B C) BACKTRACK)

A SETQ (as well as a lambda expression) must have BACKTRACK as third argument if it is to cause the establishment of a choice point. The first time this SETQ is evaluated, X will be bound to A, B, or C, and Y to the set of remaining elements. Suppose X is bound to A; then Y will be bound to (SET B C). A failure afterwards will cause a return to the backtrack point. Then X and Y will be rebound, say to B and (SET A C), and execution proceeds immediately after the SETQ.

Example: Suppose CHOICE is bound to the lambda expression (LAMBDA (SET ←X ←←Y) $X BACKTRACK) and the following program is executed.

```
(SETQ ←U 23)
(SETQ ←Z ($CHOICE (SET $U B C)))
(SETQ ←U (PLUS 2 $U))   .
```

When the application of $CHOICE to (SET $U B C) is evaluated, a backtrack point is established and Z is bound to an arbitrary element of the set. Then U is rebound to 25. If then a failure occurs in the execution of the program, control passes back to the choice point, the application of $CHOICE to (SET $U B C), the value of U is restored to 23 and another application of $CHOICE to the set is made.

Example: Suppose we execute

```
(EXISTS (CONGRESSMAN ←X))
(ASSERT (PRESIDENT $X))    .
```

Then X is bound to an arbitrary congressman, who is then asserted to be president. If a failure occurs afterwards, control returns to the EXISTS backtrack point, the system is restored so that the chosen congressman is no longer asserted to be president, and a new congressman is chosen.

C. The Goal Mechanism

One of the most widely used problem-solving techniques is the problem-decomposition method. Given a goal to achieve, first an attempt is made to find a solution directly. If no immediate solution is found,

the problem is broken down into subgoals. Each of the subgoals is now tried separately. QA4 makes it easy to use a goal/subgoal approach to solving a problem.

To encourage and simplify use of the subgoal method, QA4 provides the GOAL construct, a mechanism for activating appropriate functions without calling them by name.

1. <u>The Goal Statement</u>

The general format of the goal statement is

(GOAL <u>goal-class</u> <u>goal-expression</u> <u>indicator1</u> <u>prop1</u>

<u>indicator2</u> <u>prop2</u> ...) .

An example of a goal statement is

(GOAL $DO (STATUS LIGHTSWITCH1 ON)) .

The GOAL statement will first act as an application of the EXISTS statement to the goal expression with the given properties. When no properties are given, the default pair MODELVALUE TRUE is given. In our example a search is made in the net for the expression (STATUS LIGHTSWITCH1 ON) with indicator-property pair MODELVALUE TRUE on its property list. If the lightswitch is already on, that is, at some earlier time the assertion

(ASSERT (STATUS LIGHTSWITCH1 ON))

was made, the retrieved expression is returned as value of the GOAL statement. However, in general, no matching expression is found in the

data base. In that case a function is activated. This function should be applicable to the GOAL expression; that is, its bound variable should match that expression. In our case we would look for a function that would apply to the expression

(STATUS LIGHTSWITCH1 ON) .

To limit the number of applicable functions a further restriction is imposed, that the function be included in goal class, the first argument of the GOAL statement. The value of goal class has to be a tuple with function names as elements. These functions are the functions that will be applied in turn to the goal expression. In our example we could have assigned to $DO the value (TUPLE TURNONLIGHT SEARCHFORSWITCH). In that case the function $TURNONLIGHT would be activated. If $TURNONLIGHT's bound variable did not match (STATUS LIGHTSWITCH1 ON) or if the function failed (see the next section) somewhere in its evaluation, control would return to the GOAL and now SEARCHFORSWITCH would be activated.

It is possible for a program to be included in several goal classes. This is a way of expressing that a program is useful for achieving more than one kind of goal. A detailed example of the working of the goal mechanism is given in Section X and in the third example of Section VII-D-2.

D. Failure

When during evaluation of a program or arbitrary QA4 expression a failure is generated, the system will return to the most recent backtrack point in its history. Failure can be caused in the following ways:

- Failure of a match
- Exhaustion of all possible alternatives at a choice point
- User-invoked failure: the FAIL statement.

We will now give examples of each of these types of failure.

1. Failure of a Match

Example: (SETQ (TUPLE ←X A) (TUPLE B C)) causes the system to fail and return the most recent backtrack point.

Example: (SETQ (BAG ←X ←X) (BAG 1 2))

Example: (LAMBDA (←FOO (TUPLE A ←←X)) ($FOO $X)) will cause a failure when applied to argument

(FUNCTION1 (TUPLE B C D))

because the bound variable pattern

(←FOO (TUPLE A ←←X))

does not match the argument (FUNCTION1 (TUPLE B C D)).

2. Exhaustion of Choices

It is possible to return to a backtrack point (in QA4 jargon, "fail back") too often and exhaust the set of possible choices. The following examples illustrate this exhaustion for several QA4 constructs.

Example: (SETQ (SET ←X ←Y) (SET 1 2) BACKTRACK) .

Evaluation of this expression will establish a backtrack point and bind X to 1 and Y to 2. Failing back to the SETQ establishes the binding X ⇒ 2, Y ⇒ 1. If a failure occurs once more in the subsequent program, no new binding is possible and the statement fails.

Example: (EXISTS (TYPE ←M BOX)) .

A backtrack point will be established the first time this expression is encountered. Suppose we have asserted (TYPE BOX1 BOX), (TYPE BOX2 BOX) and (TYPE BOX3 BOX); then failing to the EXISTS will deliver successively the bindings BOX1, BOX2, and BOX3 for M. A fourth try, however, will cause a failure.

Example: (GOAL $DO (INROOM BOX1 ROOM4)) .

The programs GO1 and PUSH are put into the goal class $DO by evaluating (SETQ ←DO (TUPLE GO1 PUSH)). If no assertion (INROOM BOX1 ROOM4) has been made, programs in the goal class $DO are evoked.

Suppose GO1 is activated but the bound variable of GO1 does not match the goal expression (INROOM BOX1 ROOM4); then the other member of the goal class $DO, the function PUSH, is tried.

Suppose that the bound variable part of PUSH does match (INROOM BOX1 ROOM4) but that some failure occurs in the execution of the PUSH program. Then control returns to the GOAL statement; however, there are no other functions in the class DO, so exhaustion of the choices causes the GOAL statement to fail.

3. Explicit Failure

An explicit failure can be evoked by the statement (FAIL). The following example uses the IF statement explained in the next section.

Example:
```
(IF (GT $EFFORT 20)
    THEN (GO LABEL1)
    ELSE (FAIL))   .
```

E. Conditional Statements

A conventional IF statement is provided. A less conventional ATTEMPT statement allows us to branch on failures.

1. IF Statement

The IF statement has the general form

$$(\text{IF } e_1\ e_2\ \ldots e_n \text{ THEN } e'_1\ e'_2\ \ldots\ e'_m \text{ ELSE } e''_1\ e''_2\ \ldots\ e''_k) \quad .$$

The expressions $e_1 \ldots e_n$ are evaluated in order. If the value of the last expression e_n is FALSE the ELSE part, $e''_1, e''_2, \ldots, e''_k$, is evaluated

in order and the value of the last expression e''_k is returned as the value of the IF statement. If the value of e_n is anything other than FALSE the THEN part is evaluated, that is $e'_1, e'_2, \ldots, e'_m$ are evaluated in order and the value of the last expression e'_m is returned as value of the IF statement.

Example: In this example we will use the GO statement, which is similar to LISP's GO and will be explained in Section F-3.

```
(IF(EQ $X 3) THEN (SETQ ←FLAG SUCCESS) (PLUS $X 2)
              ELSE (SETQ ←FLAG MISSED) (GO AA))   .
```

If the value of X is 3, then FLAG will be set to SUCCESS and 5 returned as value. If the value of X is not equal to 3, FLAG will be set to MISSED and control of the program will be transferred to the statement following the label AA.

The THEN or the ELSE part of an IF statement can be omitted. If the ELSE part is omitted, and the value of e_n is FALSE, the IF statement returns the value of last evaluated expression, in this case, the value of e_n: FALSE. If the IF appears as one of a sequence of expressions, control will pass to the next statement. When the THEN part is omitted and the value of e_n is not false, the conditional returns the value of e_n.

Example: (IF ($IS BLUE BOX1) THEN ($PAINT BOX1))

($WRAP BOX1) .

If the box is blue, it will be painted as well as wrapped; otherwise, it will simply be wrapped.

2. Attempt Statement, Limiting the Scope of Failures

The ATTEMPT statement has the general form

$$(\text{ATTEMPT } e_1\ e_2\ \ldots\ e_n \text{ THEN } e'_1\ \ldots\ e'_2\ \ldots\ e'_m \text{ ELSE } e''_1\ e''_2\ \ldots\ e''_k) \quad .$$

Its evaluation is similar to that of the IF statement, with failure playing the role of the value FALSE. The expressions $e_1, e_2, \ldots, e_n$ are evaluated in turn, but if one of them generates a failure, control is passed immediately to the ELSE part. If none of the $e_1, e_2, \ldots, e_n$ fails, control is passed to the THEN part.

Example: (ATTEMPT (SETQ ←X 4)

(FAIL)

THEN 3

ELSE 5)

will bind X to 4 and return 5.

Example: (ATTEMPT (FAIL)

(SETQ ←X 4)

THEN 3

ELSE 5)

will not change the binding of X and will return 5.

The ATTEMPT statement can be used without the THEN or the ELSE part.

a. (ATTEMPT $e_1 \ldots e_n$) is very useful for protecting against unwanted returns caused by failure to backtrack points outside the ATTEMPT. A failure in the ATTEMPT will not fail back to a backtrack point outside its scope--the ATTEMPT statement will not transmit a failure--instead ATTEMPT will return value FALSE.

b. (ATTEMPT $e_1 \ldots e_n$ THEN $e'_1 \ldots e'_m$). In case of failure in $e_1 \ldots e_n$ the ATTEMPT returns with value FALSE.

c. (ATTEMPT $e_1 \ldots e_n$ ELSE $e''_1 \ldots e''_k$). If no failure occurs in $e_1 \ldots e_n$ the value of e_n is returned as value of the ATTEMPT.

Example: The following example is taken from the robot problem described in Section XI. Suppose the robot has moved and we want to update the model of the robot world accordingly. Then we have to retrieve and deny those facts that are no longer true, using such statements as the following:

(ATTEMPT (SETQ ←X (EXISTS (NEXTTO ROBOT ←Y))) THEN (DENY $X)). If an assertion of the form (NEXTTO ROBOT ←Y) is found then X is set to the retrieved expression and the THEN part of the

ATTEMPT is evaluated. Suppose the expression found was (NEXTTO ROBOT DOOR7); then DENY makes the MODELVALUE property of the expression FALSE. If no matching expression is found, the EXISTS generates a failure. The ATTEMPT returns FALSE as its value in this case.

F. The PROG Feature

1. PROG

A PROG feature similar to LISP's PROG is included. The general form of the PROG statement is

(PROG (DECLARE $var_1 \ldots var_n$) $e_1 \ldots e_n$) .

Example:

```
(PROG (DECLARE X Y)
      (SETQ ←X 4)
      (SETQ ←Y 9)
      (SETQ ←X (PLUS $X $Y))
      (RETURN $X))   .
```

Local variables are declared with the DECLARE statement. Notice that no prefixes are used for the variables.

2. RETURN

For leaving the PROG the RETURN statement is used. The general form of the RETURN statement is

(RETURN e) .

When RETURN is executed, the PROG in which it occurs will be exited with its instantiated argument _e_ as value.

```
Example:  (PROG ...

                    (RETURN (INROOM BOX1 $Z)) ...)   .
```

If Z has the value ROOM4 then (INROOM BOX1 ROOM4) will be returned as value of the PROG. RETURN may appear at any level nested inside other statements.

3. GO

Identifiers can be used as labels. The GO statement transfers control to the statement following the label provided as argument of the GO. GO may appear at any level nested inside other statements.

```
Example:  (PROG (DECLARE X TEMP)
                     - - -
                     - - -

                AA (SETQ ←X (PLUS $TEMP 4))
                     - - -
                     - - -

                   (GO AA)
                     - - - )   .
```

VII THE CONTEXT MECHANISM

A. Introduction

All properties associated with an expression can be stored and retrieved with respect to a "context," a scope for binding a variable or, more generally, a scope for assignment of properties to an expression. Under the same indicator, one expression can have different properties with respect to different contexts. The same variable can have one value with respect to one context, and another value with respect to another context. Contexts are a means for the user to create scopes for properties of expressions independently of the block and calling structure of the programs.

To make an assignment of a value to a variable or of a property to an expression with respect to a context means that the new value or property is available within the context, but that any old value or properties are still retained outside the context.

A statement that creates a new context is of the form: (CONTEXT _action_ _old-context_), where action is either PUSH or POP while old-context is one of CURRENT, GLOBAL, or $variable. To assign properties or retrieve expressions and properties with respect to a context outside of the current one, statements that handle properties of expressions are given an additional constant WRT (with respect to) and $variable as arguments. These statements include EXISTS, INSTANCES, GOAL, PUT, GET, ASSERT, DENY, and SETQ.

Example: (EXISTS (INROOM ROBOT ROOM1) WRT $C) .

B. Creating a Context

The statement (CONTEXT PUSH old-context) has as value a new context. If we use GLOBAL for old-context the newly created context has all top-level bindings available. When CURRENT is used for old-context, all bindings existing at the moment of evaluation of the context statement are available. If old-context is some context $C, all the bindings available in context $C will be available in the new context also.

Example: Evaluating

(SETQ ←NEWCONTEXT (CONTEXT PUSH GLOBAL))

will assign a new context to the variable NEWCONTEXT. This context has all top-level bindings of the QA4 system available. Note that this assignment merely names a new context and does not change the current context.

C. Building a Tree of Contexts

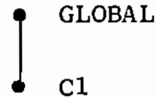
We can build a tree of contexts by using $variables as second arguments (old-context) in the CONTEXT statement. The following example demonstrates how such a tree structure is built and shows what bindings are available at each node of a tree.

Assume we have the top-level bindings

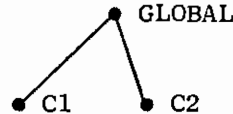
(SETQ ←X 2)

(ASSERT (SIGNAL RED)) .

Then (SETQ ←C1 (CONTEXT PUSH GLOBAL)) will create a new context in which X has value 2 and (SIGNAL RED) is asserted TRUE. We could picture this as



Evaluating (SETQ ←C2 (CONTEXT PUSH GLOBAL)) will return a context identical to C1. The initial context now has two descendants. In a diagram:



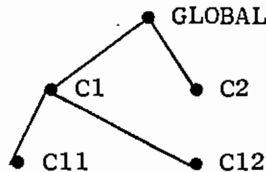
If we want to assert the expression (SIGNAL RED) to be false in context C1, we can evaluate (DENY (SIGNAL RED) WRT $C1). Notice that in both the global context and context C2 the expression (SIGNAL RED) still has value TRUE.

Evaluation of

(SETQ ←C11 (CONTEXT PUSH $C1))

and (SETQ ←C12 (CONTEXT PUSH $C1))

will create two descendant nodes of node C1. The tree now looks like



In both contexts C11 and C12, the expression (SIGNAL RED) will have value FALSE, while X has value 2. Thus, while (EXISTS (SIGNAL RED) WRT $C12) will fail, (EXISTS (SIGNAL RED) WRT $C2) will not fail.

D. Creating a Context by Referring to Foregoing Nodes in a Context Tree

We can create a new context by "popping" contexts. The value of (CONTEXT POP $CT) is the context of which CT was the descendant. In our example

(CONTEXT POP $C11)

will have as value context C1, while

(CONTEXT POP $C1)

will have context GLOBAL as value.

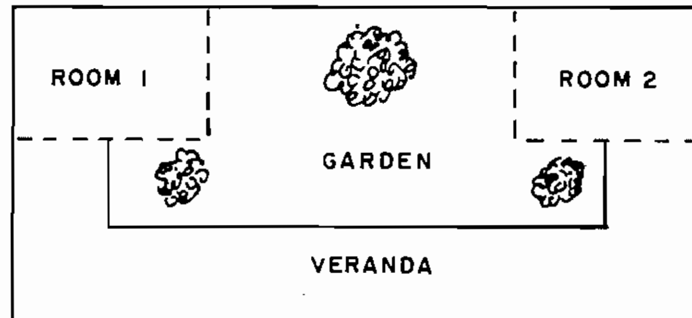
E. An Example

We will now illustrate the use of contexts with a simple example. In this example we derive a goal under hypothetical assumptions. Suppose we want to derive goal g under the assumption that the expression exp has MODELVALUE TRUE. We express this in QA4 as

```
(SETQ ←C1 (CONTEXT PUSH CURRENT)
(ASSERT exp WRT $C1)
(GOAL $PROVE g WRT $C1)    .
```

The assertion of exp is made only with respect to the context C1; exp will not have MODELVALUE TRUE outside C1. All changes in the data base made by the derivation of the GOAL are visible only by referring to the created context C1. In this way the user is given explicit control of

a mechanism that is usually tied to the block or calling structure of the program. In our example, we have a simple world with a robot whose environment consists of two rooms, a garden, and a roofed veranda



The robot is not waterproof and so uses the veranda to go from one room to another whenever it rains but goes through the garden when it is sunny. A plan is needed to move the robot between the rooms. This plan should work in both rainy and sunny weather. Thus, it must be a conditional plan. The given program is not the simplest solution to this specific problem, but is intended as a demonstration of the use of contexts for derivation of goals under hypothetical assumptions.

In the initial state we assert the robot to be in ROOM1 by entering

```
(ASSERT (INROOM ROBOT ROOM1))   .
```

At the time of the planning it is uncertain how the weather will be when the plan is used, so we assert

```
(ASSERT (UNCERTAIN (STATUS WEATHER RAINY)
                   (STATUS WEATHER SUNNY)))   .
```

We will use three functions or operators: GOROOM, GOTHRUGARDEN, and GOTHRUVERANDA.

GOROOM returns a plan for going between rooms. This plan will have the robot take the appropriate route according to the weather. The version of GOROOM presented here is simplified and somewhat artificial; however, it is meant merely to illustrate the use of the context mechanism.

The operators GOTHRUGARDEN and GOTHRUVERANDA do the planning for moving the robot from one room to the other through the garden and veranda respectively.

The GOTHRUGARDEN operator has as the precondition for its use that the weather be sunny, while the GOTHRUVERANDA operator has the precondition that the weather be rainy.

The operators are:

GOROOM, with definition

```
    (LAMBDA (INROOM ROBOT ←ROOM)

                                        (COMMENT this function applies to
                                        expressions of the form (INROOM ROBOT
                                        exp), where exp can be any expression.)

    (PROG (DECLARE X Y PLAN1 PLAN2)
        (EXISTS (UNCERTAIN
                    (STATUS WEATHER ←X)
                    (STATUS WEATHER ←Y)))

                                        (COMMENT retrieve the given expression
                                        from the data base and find X and Y; X
                                        and Y are complementary.)
```

```
(SETQ ←C1 (CONTEXT PUSH CURRENT))

                          (COMMENT create a new context C1 which
                          contains all current bindings.  Note
                          that the current context is not changed;
                          this assignment merely names a new con-
                          text.)

(ASSERT (STATUS WEATHER $X) WRT $C1)

                          (COMMENT assert with respect to C1
                          that the status of the weather is the
                          value of X.)

(SETQ ←PLAN1
    (GOAL $MOVE (INROOM ROBOT $ROOM) WRT $C1))

                          (COMMENT make a plan for moving the
                          robot which takes into account that
                          the weather has the previously
                          asserted value.)

(SETQ ←C2 (CONTEXT PUSH CURRENT))

                          (COMMENT create a second context C2
                          and repeat the planning under a dif-
                          ferent assumption.  Note that all
                          changes in the data base made by the
                          first planning are ignored by the use
                          of a new context.)

(ASSERT (STATUS WEATHER $Y WRT $C2))
(SETQ ←PLAN2
    (GOAL $MOVE (INROOM ROBOT $ROOM) WRT $C2))

(RETURN
    ('(IF (EXISTS (STATUS WEATHER $X))
          THEN $$PLAN1
          ELSE
          (IF (EXISTS (STATUS WEATHER $Y))
          THEN $$PLAN2)))))

                          (COMMENT combine the two plans in a
                          program with a conditional.  X and Y
                          are instantiated and the elements of
                          the values of PLAN1 and PLAN2 are
                          inserted at the same level as THEN
                          and ELSE.
```

GOTHRUGARDEN, with definition

```
(LAMBDA (INROOM ROBOT ←Y)
    (PROG (DECLARE Z)
        (EXISTS (STATUS WEATHER SUNNY))
        (DENY (EXISTS (= (INROOM ROBOT ←Z))))
        (ASSERT (INROOM ROBOT $Y))
        (RETURN ('(GOTHRUGARDEN $Y)))))
```

GOTHRUVERANDA, with definition

```
(LAMBDA (INROOM ROBOT ←Y)
    (PROG (DECLARE Z)
        (EXISTS (STATUS WEATHER RAINY))
        (DENY (= (EXISTS (INROOM ROBOT ←Z))))
        (ASSERT (INROOM ROBOT $Y))
        (RETURN ('(GOTHRUVERANDA $Y)))))    .
```

The initial goal statement which starts the problem-solving process is

```
(GOAL $GO (INROOM ROBOT ROOM2))    .
```

The robot is initially in ROOM1, this fact is expressed by making the assertion

```
(ASSERT (INROOM ROBOT ROOM1))    .
```

We will assign the operators to appropriate goal classes

```
(SETQQ ←MOVE (TUPLE $GOTHRUGARDEN $GOTHRUVERANDA))
(SETQQ ←GO (TUPLE $GOROOM))    .
```

The operator GOROOM sets up two new contexts, C1 and C2. In one of these contexts, say C1, (STATUS WEATHER SUNNY) is asserted, and in the other, C2, (STATUS WEATHER RAINY) is asserted. In each of these contexts the goal (INROOM ROBOT $ROOM) has to be achieved, so that the

plan will take both weather conditions into account. In context C1, GOTHRUVERANDA will fail and GOTHRUGARDEN will succeed, while in context C2, GOTHRUGARDEN will fail and GOTHRUVERANDA will succeed. GOROOM puts the two resulting plans together into a single plan that will have the robot check the weather and behave accordingly.

VIII A ROBOT SYSTEM

A. Introduction

The remainder of this chapter illustrates the application of QA4 to some simple robot problems.

QA4 has been designed with a specific problem-solving philosophy, which it subtly encourages its users to adopt, and which is an outgrowth of our experience with its antecedent, QA3. QA3 contained an axiom-based theorem prover, which we attempted to use for general-purpose problem solving. However, all knowledge had to be stored in the declarative form of logical axioms, with no indication as to its use. When a large number of facts were known, the knowledge could not be used effectively. The system became swamped with irrelevant inferences, even when supplied with several sophisticated syntactic strategies.

In contrast, QA4 can store information in an imperative form, as a program (Winograd 1971). This makes it possible to store strategic advice locally rather than globally. In giving information to the system, we can tell it how that information is to be used. Strategies tend to be semantic rather than syntactic. We are concerned more with what an expression means than with how long it is. QA4 programs are intended to rely on an abundance of know-how rather than a large search in finding a solution. We expect our problem solver to make few poor choices, and we try to give it all the information at our disposal to restrict these choices.

B. The Robot Problems

We will now examine the kind of knowledge we expect a robot planner to have, and the class of problems we expect it to be able to solve. We will consider some problems of a type recently approached by the SRI robot (Fikes 1971) (Raphael 1969). We will then be better able to discuss the application of QA4 to this domain and the merits of the QA4 approach.

We envision a world consisting of several rooms and a corridor, connected by doorways. There are boxes and other objects in some of the rooms, and there are switches that control the lights. The robot can move freely around the floor, can pass between the rooms, can see and recognize the objects, can push all the objects, and can climb up onto the boxes. If the robot is on top of a correctly positioned box, it can switch the light on and off.*

The first problem faced by the robot is to turn on the light in one of the rooms. To solve this problem it must go to one of the boxes, push the box next to the lightswitch, climb up on the box, and turn the switch.

We supply the problem solver with a MODEL or representation of the world, which includes the arrangement of the rooms and the positions of and relationships between the objects. Furthermore, corresponding to

---

*Actually, the robot that exists at SRI can neither climb boxes nor turn switches.

each action the robot can take, we supply an OPERATOR, whose effect is to alter the model to reflect the changes the robot's action makes on the world. Each operator has PRECONDITIONS, requirements that must be satisfied before it is applied.

For example, the PUSHTO operator corresponds to the robot's action of pushing a box. It changes the model by changing the location of the robot and the location of the box. Its precondition is that the robot be next to the box before the operator is applied.

The GOAL of the problem is a set of conditions that we want the model to satisfy. For example, in the problem of turning on the light, we require that the light be on when the task is completed. The problem of planning, then, amounts to the problem of finding a sequence of operators that, when applied to the initial world model, will yield a new model that will satisfy the goal condition. If the robot then executes the corresponding sequence of actions it will, presumably, have solved the problem.

Let us suppose that the problem solver works backwards from its goal in its search for a solution. It finds an operator whose effect is to change the model in such a way that the goal condition is satisfied. However, the preconditions of that operator might not be true in the initial model. These preconditions then become subgoals, and the problem solver seeks out operators whose effect is to make the

subgoals true. This process continues until all preconditions of each operator in the solution sequence are true in the model in which that operator is applied, and thus, in particular, the preconditions of the first operator in the plan are true in the initial model.

IX THE SOLUTION OF THE PROBLEM OF TURNING ON A LIGHT

A. <u>The Framework</u>

Let us examine within this framework the complete solution of the problem of turning on a light. We assume that, in the initial model, the lightswitch, the robot, and at least one box are all in the same room. We assume that the problem solver can apply a set of operators that includes the following: TURNONLIGHT, CLIMBONBOX, PUSHTO, and GOTO, which correspond to the actions necessary to turn on the light. The goal is that the status of the light be ON. The operator TURNONLIGHT has the effect of making this condition true. However, the preconditions of this operator, that the box be next to the lightswitch and the robot be on top of the box, are not true in the initial model. These preconditions therefore become new subgoals. For the robot to be on top of the box, it suffices to have to have applied the CLIMBONBOX operator. However, this operator has the precondition that the robot be next to the box; this precondition becomes a new subgoal.

Both this subgoal and the unachieved precondition of the TURNONLIGHT operator, that the box be next to the lightswitch, are achieved by the PUSHTO operator, which can move a box anywhere in the room. However, the PUSHTO operator still has the precondition that the robot be next to the box. This new subgoal can be achieved by the GOTO operator, which can move the robot anywhere around the room. The only precondition of the

GOTO operator is that the robot be in the same room as its destination, but this condition is satisfied in our initial model, since the robot and the box are assumed to be in the same room. Thus, a solution has been found, the sequence: GOTO the box, PUSHTO the box next to the lightswitch, CLIMBONBOX, and TURNONLIGHT.

The QA4 solution to the robot problems is a direct translation of the approach of the STRIPS problem-solving system, which uses the above framework. STRIPS is the problem solver that does the planning for the SRI robot. The solution of the first three problems approached by STRIPS was the first exercise for the QA4 language. The operators were encoded in the QA4 language, and the model was expressed as a sequence of QA4 statements. This package of information, with no further supervision or strategy, sufficed for the solution of the three sample problems. Finding the plans amounted to evaluating the goals expressed in the QA4 language. The solutions were found quickly and with no more search than necessary. More significantly, the operator descriptions were written quickly and are concise and fairly readable.

B. The STRIPS Representation

For STRIPS, the model is a set of sentences in first-order logic; the preconditions of an operator are also expressed as a set of first-order sentences. The description of the operator itself is restricted to a rather rigid format: There is a delete list, a set of sentences

to be deleted from the old model, and the add list, a set of sentences to be added to the new model. The delete list expresses facts that may have been true before the action is performed but that will not be true afterwards. In STRIPS, the TURNONLIGHT operator, for instance, is described as follows: Its preconditions are that the robot be on the box and that the box be next to the lightswitch. It deletes from the model the fact that the light is OFF, and it adds to the model the fact that the status of the light is ON. In STRIPS, the strategy for selecting and forming sequences of operators is embodied in a large LISP program. The applicability of operators and the differences between states are frequently determined by a general-purpose first-order theorem prover, and the operators themselves are coded in a special-purpose Markov Algorithm language. In QA4, all these elements of the problem-solving system can be handled within a single formalism. We can use the full power of the QA4 programming language to construct the operator description. To describe the operators we have discussed above, we follow the STRIPS format rather closely. For more complex operators and plans, we may make use of more of the language features, as we shall see below.

C. The QA4 Representation

We will now look at the QA4 program for the TURNONLIGHT operator;* the reader can thus become familiar with the flavor and some of the features of QA4 without having to read a general description:

```
(LAMBDA (STATUS ←M ON)
    (PROG (DECLARE N)
        (EXISTS (TYPE $M LIGHTSWITCH))
        (EXISTS (TYPE ←N BOX))
        (GOAL $DO (NEXTTO $N $M))
        (GOAL $DO (ON ROBOT $N))
        ($DELETE (' (STATUS $M OFF)))
        (ASSERT (STATUS $M ON))
        ($BUILD (' (:$TURNONLIGHTACTION $M)))) .
```

First, we summarize the action of this operator on the model: It selects a box and asks that the box be next to the lightswitch and that the robot be on top of the box. It then turns the light on and it adds the turning on the switch to the sequence of actions to be executed by the robot.

The reader will note how concise and readable the QA4 representation of operators is. Now we will examine the TURNONLIGHT operator in more detail, to see what it does and the constructs it uses.

---

*The QA4 programs for the other operators, the precise formulation of the lightswitch problem, and the tracing of the solution of that problem, are included in Appendix I.

1. The Pattern

The program has a LISP-like appearance, but it is evaluated by a special interpreter. In place of a bound variable list, it has a pattern (STATUS ←M ON). This pattern serves as a relevancy test for application of the function. An operator will be applied only to goals that match its bound variable pattern.* This operator will be applied only when something is to be turned on. In STRIPS, the add list serves the same function as the pattern. However, in QA4 the relevancy test is distinct from the commands that change in the model.

All variables in QA4 have prefixes; the prefix ← of the variable M means that the pattern element ←M will match any expression, and M will then be bound to that expression. The other two pattern elements, STATUS and ON, have no prefixes: They are constants and will match only other instances of themselves.

For the following example we shall assume that we want to turn on LIGHTSWITCH1; our goal, therefore, is (STATUS LIGHTSWITCH1 ON). The pattern of the TURNONLIGHT operator matches this goal, binding M to LIGHTSWITCH1.

Patterns play many roles in QA4; they may appear on the left side of assignment statements and in data base queries. The ability to have a pattern as the bound variable part of a function gives us a

---

*In the primer we use the term "bound variable part."

concise notation for naming substructures of complex arguments. It also gives us a flexible alternative to the conventional function-calling mechanism, as we shall see.

2. Searching the Data Base

The program must first be sure that the value of M is a lightswitch. This is one of the preconditions of the operator. The statement (EXISTS (TYPE $M LIGHTSWITCH)) searches for instances of the pattern (TYPE $M LIGHTSWITCH) that have been declared TRUE in the data base.

The $ prefix of the variable M means that $M will match only instances of the value of M; M will never be rebound by this match. Thus, in our example we look only for the expression (TYPE LIGHTSWITCH1 LIGHTSWITCH) in the data base.

Unless otherwise specified, the EXISTS statement also checks that this expression has been declared true. Expressions have values; to declare an expression true, we use the ASSERT statement. This construct sets the value of its argument expression to TRUE. This value is stored in the property list of the expression. In QA4, the model is the set of expressions with value TRUE. For our example, we assume that the user has input (ASSERT (TYPE LIGHTSWITCH1 LIGHTSWITCH)) before attempting the problem. Thus, the fact that LIGHTSWITCH1 is a lightswitch is included in the model.

The EXISTS statement will cause a failure if no suitable expression is found in the data base. A failure initiates backtracking. Control passes back to the last point at which a choice was made, and another alternative is selected. Much of the power of the QA4 language lies in its implicit backtracking, which relieves the programmer of much of the bookkeeping responsibility.

3. Choosing a Box

The operator uses another EXISTS statement to choose a box to use as a footstool: (EXISTS (TYPE ←N BOX)) searches the data base for an expression of the form (TYPE ←N BOX) whose value is TRUE. That such a box exists is one of the preconditions of the operator. Note that here the variable has prefix ←, so there is a class of expressions the pattern will match, and the variable N will be bound by the matching process. We will assume that (TYPE BOX1 BOX) has been asserted to be true, and that N is bound to BOX1.

If for some reason the operator is unable to use BOX1, a failure will occur. Control will pass back to the EXISTS statement, which will then select another box.

4. Moving the Box

The operator now insists as one of its preconditions that the chosen box be next to the lightswitch. For this purpose it uses the GOAL construct, a mechanism for activating appropriate functions

without calling them by name. To move the box, the operator uses (GOAL $DO (NEXTTO $N $M)).

The GOAL first acts as an EXISTS statement: It checks to see whether (NEXTTO $N $M), that is, (NEXTTO BOX1 LIGHTSWITCH1), is in the data base. If BOX1 is already next to the lightswitch, the goal has already been achieved. However, in general, it will be necessary to move the box by using other operators. In other words, the precondition is established as a subgoal.

Every operator has a bound variable pattern and a "goal class," a user-defined heuristic operator partition. A GOAL statement specifies an expression and a goal class. In these problems there are two goal classes, DO and GO. The operators in the GO class are those that simply move the robot around on the floor: for example, GOTO. The operators that move objects or that cause the robot to leave the floor are in the DO class: PUSHTO, CLIMBONBOX, and TURNONLIGHT.

An operator can only be applied to a goal if it belongs to the goal class specified by the GOAL statement. In our example the goal class is DO. Therefore, the only operators that can be applied are PUSHTO, CLIMBONBOX, and TURNONLIGHT.

Each of the operators has a bound variable pattern. To be applied to a goal it is not sufficient that the operator belong to the specified goal class; it is also necessary that the bound variable

pattern of the operator match the expression specified by the goal statement. In this case the bound variable pattern of the PUSHTO operator, (NEXTTO ←M ←N), matches the goal expression (NEXTTO BOX1 LIGHTSWITCH1), with M bound. Therefore, the PUSHTO operator is activated.

We will be somewhat more sketchy about the operation of the PUSHTO operator, since our aim is to focus attention on the TURNONLIGHT operator. The PUSHTO operator establishes another subgoal, (NEXTTO ROBOT BOX1), with goal class GO. This goal activates the operator GOTO, which succeeds without establishing any further subgoals.

The GOAL mechanism is powerful because we need not know in advance which functions it will activate; that choice depends on the form of the argument. The relevant operators come forward at the appropriate time.

The TURNONLIGHT operator requires not only that the box be next to the lightswitch, but also that the robot be on top of the box, before it can turn on the light. This precondition is described as (GOAL $DO (ON ROBOT $N)). Of the operators in the DO class, the only one whose bound variable pattern matches the goal expression is CLIMBONBOX, with pattern (ON ROBOT ←M), so this operator is applied.

The preconditions of this operator, including the requirement that the robot be next to BOX1, are already satisfied in the model. Thus, the operator can report a quick success.

The remaining statements effect the appropriate changes in the model. The statement ($DELETE (' (STATUS $M OFF))) corresponds to the specification of the delete list in the STRIPS description of the operator. There is a $ prefix on the DELETE function because this function is a user-defined function: Its definition is the value of the variable DELETE. The statement (ASSERT (STATUS $M ON)) represents the add list of the operator. The final statement ($BUILD (' (:$TURNONLIGHT ACTION $M))), simply adds the action of turning on the light to the planned sequence of actions to be carried out by the robot.

D. Design Philosophy Revisited

Although our operators are as concise as those of STRIPS, we have given them a certain amount of strategic information. For example, in TURNONLIGHT we tell the system that the box must be brought up to the lightswitch BEFORE the robot mounts the box, while in STRIPS the same preconditions are unordered, so the planner may investigate the ill-advised possibility of climbing the box first and moving it later. STRIPS could have been given ordered preconditions, but its designers were more interested in the behavior of the problem solver when it had

to discover the best ordering by itself. The decision about how many hints to give the operators is, we feel, primarily a matter of taste. For QA4 we prefer to give the operators as much information as possible, and risk the charge of dealing with problems on an ad hoc basis. We feel that this is the only way that our programs will solve interesting problems.

Although STRIPS does not rely as heavily on axiom-based theorem proving as QA3 does, it still uses a theorem prover for such purposes as determining whether an operator is relevant or applicable, tasks that QA4 accomplishes by using pattern matching. As STRIPS has shown us, the theorem proving involved in such processes is quite straightforward, and a pattern matcher seems to be a more appropriate tool here than a full-fledged theorem prover.

We also applied the system to the two problems from the STRIPS paper (Fikes 1971). In these problems the robot is envisioned to be in a building with several rooms and a corridor. It is asked first to push together the three boxes in one of the rooms, and second to find its way from one of the rooms to another one. The QA4 system solved these problems also, and it made no mistaken choices.

These problems are, of course, particularly simple. Had they been sufficiently complicated, any QA4 program would have to do some searching in trying to find a solution. In that case, we would have had to write our operators in a somewhat different way.

E. Other Features and Applications

These problems did not use many of the features of QA4 that we feel would be valuable for more complex problems and in other problem domains. For example, STRIPS plans are always linear sequences of operators; plans never include branches that prepare for various contingencies. There is no mechanism for considering alternative world models in a single plan. The construction of conditional plans is facilitated by the "context mechanism" of QA4, which allows us to store alternative hypotheses under distinct contexts without confusion.

STRIPS plans also have no loops; an action in a STRIPS plan can be repeated only a prespecified number of times. However, the fact that QA4 plans are programs that admit both iteration and recursion opens the possibility of writing plans with repeated actions.

Since QA4 is successful in writing robot plans with loops, it will probably be equally effective at the synthesis of computer programs. Assembly code programs, in particular, are strikingly similar to robot plans: Computer instructions are analogous to operators, and whereas for robot plans we model the world, for computer programs we model the state of the registers of the machine. QA4 has already been successful at producing simple straight-line assembly code programs. More general theorem-proving ability would enable QA4 to construct programs in other languages, including QA4 itself.

We have applied QA4 to the verification of existing programs. For this application we need considerable sophistication in formula manipulation and the handling of arithmetic relations. We have introduced some new data types, sets, and bags (bags are like sets, but may have several instances of the same element), which simplify many arithmetic problems. For example, a stumbling block of earlier deductive systems has been the equality relation, which has required either a plethora of new axioms or a slightly less clumsy new rule of inference in order to describe its properties. In QA4 we simply place expressions known to be equal in the same set; the symmetric, reflexive, and transitive laws then follow from the properties of sets, and need not be stated explicitly.

A similar technique simplifies the description of commutative functions of n arguments, such as plus and times. We make these arguments bags rather than n-tuples. Then the commutative law for addition need no longer be mentioned, since bags, like sets, are unordered.

Chapter Three

PLAN SYNTHESIZERS

CHAPTER THREE--PLAN SYNTHESIZERS

I INTRODUCTION

In this chapter we present some QA4 programs that solve robot planning problems. Each program is a simplified planner for a specific problem area of robot planning. The programs are overly simplified, and avoid the detail that would be necessary if they were joined together in a large, unified robot planner. Singly, however, each demonstrates ways the features of the QA4 language can be used to construct problem solvers based on intuitive reasoning.

We have chosen robot problems exclusively because of their highly intuitive nature. The subjects of these prototype problem solvers, however, arise in any area of automatic program synthesis. Programs or plans for either computers or robots must contain branches, loops, and the other structures dealt with in these sample automatic planners.

II CONDITIONAL PLANS

A. Predicates and Actions

Suppose that we have a set of related predicates, P1, P2, ..., Pn, whose values are unknown at the time a plan is generated, are testable at the time the plan is executed, and are disjunctively true. For program synthesis problems, the predicates may be simple relations on subroutine arguments such as $X < 0$. For robot planning, the predicates could be either relations between objects in the world such as LEFT-OF(BOX1,BOX2) or predicates on single objects of the world such as OPEN(DOOR) or RAINY. At least one predicate of the set must always be true. For example, if RAINY and SUNNY are the two ways the robot classifies days, then for the robot it is always either RAINY or SUNNY.

Our problem is to formulate a plan that satisfies a goal, even though different actions must be taken depending upon the particular predicate that is true at the time the plan is executed. Suppose, for example, our goal is to have fun, and there are two ways: If it is rainy we can go to a movie, and if it is sunny we can go to the beach. The plan would be either

(IF RAINY THEN MOVIE ELSE BEACH)

or

(IF SUNNY THEN BEACH ELSE MOVIE) .

In general, if the actions are A1, A2, ..., An, then the form of the plan should be

```
(IF P1 THEN A1 ELSE
    IF P2 THEN A2 ELSE
        ...
            IF P(n-1) THEN A(n-1) ELSE An)   .
```

Notice that the last predicate need not be tested, for the only case in which it would be tested is if all the others were false; but in that case we know it must be true.

B. Goal Satisfaction

The key to our solution will be a program that creates a new context with fewer uncertain predicates than the current context and plans for the goal in that new context. But before we study it in detail, let us assume we have such a program, say ALTPLAN, and construct a planner around it. ALTPLAN accepts as arguments a goal and a condition and takes the following actions:

- If the condition is true in the current model, then ALTPLAN will return notification that no alternate plan should be constructed. For example, the condition may be true because a more global program has assumed it is true. Thus the predicate is already known, and a branching plan does not have to be created at tnis point. The robot could be outside when the

subgoal HAVEFUN arises. Since it would never go outside in the rain, SUNNY would be assumed to be true, and it could go directly to the beach.

- If the condition is uncertain, then ALTPLAN will construct a new model where the condition is false, solve the problem in that new model, and return the plan.

C. MOVIE and BEACH

Using ALTPLAN we may construct simple programs to work on the goal. Our MOVIE program that satisfies the goal HAVEFUN is

```
[RPAQQ MOVIE
    (LAMBDA HAVEFUN
        (PROG (DECLARE ALT)
            (SETQ ←ALT ($ALTPLAN RAINY HAVEFUN))
            (IF (EQUAL (TUPLE)
                       $ALT)
                  THEN
                  (RETURN MOVIE)
                  ELSE
                  (RETURN (' (IF RAINY THEN MOVIE ELSE $ALT]   .
```

This program first calls ALTPLAN and stores the alternative plan in the variable ALT. If ALTPLAN returned the empty tuple (TUPLE), then the condition was true, so MOVIE returns the plan MOVIE. Otherwise, ALTPLAN solved the problem under the assumption that RAINY was false, so our

program returns the plan

(IF RAINY THEN MOVIE ELSE $ALT) .

The operator that formulates a plan to go to the beach is the same except for the condition and the action.

```
[PRAQQ BEACH
     (LAMBDA HAVEFUN
          (PROG (DECLARE ALT)
               (SETQ ←ALT ($ALTPLAN SUNNY HAVEFUN))
               (IF (EQUAL (TUPLE)
                          ALT)
                    THEN
                    (RETURN BEACH)
                    ELSE
                    (RETURN (' (IF SUNNY THEN BEACH ELSE $ALT]   .
```

In a large robot system, we would not expect this much similarity in detail, although the approach would always be the same.

D. ALTPLAN

Now let us examine ALTPLAN in more detail.

```
[RPAQQ ALTPLAN (LAMBDA (TUPLE ←CONDITION ←GOAL)
     (PROG (DECLARE NC Y ALT Z)
          (ATTEMPT (EXISTS $CONDITION)
                    THEN
                    (RETURN (TUPLE)))
          (EXISTS (UNCERTAIN (SET $CONDITION ←←Y)))
          (SETQ ←NC (CONTEXT PUSH CURRENT))
          (DENY (UNCERTAIN (SET $CONDITION $$Y))
              WRT $NC)
          (DENY $CONDITION
              WRT $NC)
          (ATTEMPT (SETQ (SET ←Z)
                    $Y)
                    THEN
                    (ASSERT $Z WRT $NC)
                    ELSE
                    (ASSERT (UNCERTAIN $Y)
                                    WRT $NC))
          (SETQ ←ALT (GOAL $GOALCLASS $GOAL WRT $NC))
          (RETURN $ALT]
```

The first ATTEMPT statement checks the condition in the model and returns the empty tuple as the notification that there should be no alternative plan.

The EXISTS statement retrieves the set of uncertain conditions and binds the variable Y to the set consisting of all the uncertain conditions except the condition currently under consideration. Next ALTPLAN creates a new context, NC, and then denies the set of uncertain conditions in this context. This denial is done because the context NC automatically assumes all the assertions from its parent context, but in NC, we will assume that CONDITION is FALSE, and thus is not uncertain.

The final ATTEMPT statement of ALTPLAN checks to see if there is only one uncertain condition: If so, it asserts the condition in the new context; if not, it asserts that the smaller set of conditions is uncertain.

Finally, ALTPLAN solves the problem in the context NC and returns the plan. The WRT clause on the GOAL statement has a special effect on all future model reference statements such as EXISTS and ASSERT. It makes them operate in the context NC, until, of course, either another GOAL with a WRT is established, or this current GOAL is solved. The solution trace demonstrates this simultaneous establishment of GOALS and contexts.

E. <u>(GOAL $HAVEFUN HAVEFUN)</u>

For our example, the GOAL will first call BEACH. This will immediately call ALTPLAN who will simplify the UNCERTAIN condition and reestablish the GOAL. BEACH is called again, and it calls ALTPLAN again. This time ALTPLAN fails, causing the second call on BEACH to

fail. Thus the GOAL interpreter moves on to MOVIE. MOVIE calls ALTPLAN, who discovers the condition is TRUE. MOVIE returns the plan MOVIE as the value of the GOAL. ALTPLAN passes the plan back to BEACH, who embeds it in a larger, conditional plan.

The following is a more detailed trace of the conditional plan synthesizer. The numbers refer to the following automatic trace.

We begin with the assertion (UNCERTAIN (SET RAINY SUNNY)) (1) and the goal (2). The system first checks to see if HAVEFUN is TRUE in the model. Since it is not TRUE, a FAILURE is generated (4). This was the EXISTS portion of a goal interpretation. Continuing to work on the goal, the interpreter applies the program BEACH to the constant HAVEFUN (5,6).

BEACH immediately calls ALTPLAN (7), which binds the variables GOAL(8) and CONDITION (9). Next, ALTPLAN checks CONDITION (10), which is now SUNNY, discovers it is not TRUE and FAILS (11). It then goes on to extract SUNNY from the set of uncertainties (12), and set the variable Y to the set of those predicates remaining uncertain (13). In this case, Y is set to (SET RAINY). ALTPLAN creates a new context, NC (14); denies (UNCERTAIN (SET SUNNY RAINY)) (15); and denies SUNNY (16) in the new context. It now notices that only the condition RAINY remains (17); asserts RAINY in the new context (18); and reestablishes the original goal, HAVEFUN, with the original GOALCLASS, (TUPLE BEACH MOVIE), but with respect to the new context (19).

HAVEFUN is not TRUE, so the EXISTS portion of the goal fails (21), and BEACH is called again (22). BEACH immediately calls ALTPLAN (23), and the arguments are bound (24,25). The condition SUNNY is checked (26), and is not TRUE because this is a recursive call on BEACH and SUNNY is, in fact, FALSE. Next an attempt is made to find the uncertain set (28). Since there is none in this context, the EXISTS fails (29), causing BEACH to fail, and thus causing the goal interpreter to proceed to the next program in the GOALCLASS.

MOVIE is called (30) and ALTPLAN is immediately called (31). The arguments are bound (32,33). The condition RAINY is checked (34). Since the EXISTS does not have a WRT value, the model query is made with respect to the context established by the last GOAL--GOAL (19) and context NC. RAINY was asserted (18) in this context and is TRUE. ALTPLAN proceeds to return the empty tuple (35). Since ALTPLAN returned the empty tuple, this call to MOVIE (30) returns the constant MOVIE.

The goal statement (19) is now complete, so the variable ALT in ALTPLAN is now set to the value of the GOAL statement (36)--namely MOVIE. ALTPLAN returns the variable and thus returns from the initial call in BEACH (6). The variable ALT in BEACH is set to the returned plan (37). Since the plan is not the empty tuple, BEACH returns the conditional plan (38).

CONDITIONAL PLAN

```
1.  ←! (ASSERT (UNCERTAIN (SET RAINY SUNNY)))
2.  ←! (GOAL $HAVEFUN HAVEFUN)
3.  (GOAL $HAVEFUN HAVEFUN)
4.  SETVALUE GOALCLASS (TUPLE BEACH MOVIE)
5.  FAILURE
6.     LAMBDA BEACH
7.       LAMBDA ALTPLAN
8.       SETVALUE GOAL HAVEFUN
9.       SETVALUE CONDITION SUNNY
10.      (EXISTS $CONDITION)
11.      FAILURE
12.      (EXISTS (UNCERTAIN (SET $CONDITION ←←Y)))
13.      SETVALUE Y (SET RAINY)
14.      SETVALUE NC (CONTEXT (5 4 3 2 1 0) 5 4 3 2 1 0)
15.      (DENY (UNCERTAIN (SET $CONDITION $$Y) WRT $NC)
16.      (DENY $CONDITION WRT $NC)
17.      SETVALUE Z RAINY
18.      (ASSERT $Z WRT $NC)
19.      (GOAL $GOALCLASS $GOAL WRT $NC)
20.      SETVALUE GOALCLASS (TUPLE BEACH MOVIE)
21.      FAILURE
22.         LAMBDA BEACH
23.            LAMBDA ALTPLAN
24.            SETVALUE GOAL HAVEFUN
25.            SETVALUE CONDITION SUNNY
26.            (EXISTS $CONDITION)
```

```
27.               FAILURE
28.               (EXISTS (UNCERTAIN (SET $CONDITION ←Y)))
29.               FAILURE
30.            LAMBDA MOVIE
31.               LAMBDA ALTPLAN
32.               SETVALUE GOAL HAVEFUN
33.               SETVALUE CONDITION RAINY
34.               (EXISTS $CONDITION)
35.            SETVALUE ALT (TUPLE)
36.         (SETVALUE ALT MOVIE)
37.      (SETVALUE ALT MOVIE)
38.   (' (IF SUNNY THEN BEACH ELSE MOVIE))
```

III INFORMATION GATHERING

A. Operators

As can be seen in the TURNONLIGHT problem of Chapter Two, the robot planners of our examples are organized around "operators." The operators are programs that work to satisfy goals. When they succeed, they update the model and return a plan that, when executed, will operate on the world much the way the program operates on the model. Sometimes the plan depends on information that the robot gathers during execution. Take, for example, the OPENDOOR operator. To open a door, the robot must go to the door, check the status of the door, and if it is closed, open it. Thus the operator could always build the three-step plan:

```
(GOTO DOOR)
(CHECKDOOR DOOR)
(IF (NOT (OPEN DOOR)) THEN (OPENDOOR DOOR))   .
```

If, however, this plan is a portion of a larger plan, the robot may have just moved to the door, or it may have already checked the door but have just moved away from it. If states such as these arise during planning where the model indicates some of the steps for OPENDOOR may be omitted, then the planner should omit them.

To build just the needed steps into the plan, we will construct the operator in such a way that checking the preconditions will return a plan that assures they will be true when the plan is executed. Then

the operator will update the model and return a plan composed of the plan for the preconditions together with the final steps for the operator. But first, if the door is not open in the model, we simply return the plan (OPENDOOR $DOOR). Our OPENDOOR operator has just these four steps.

```
[RPAQQ OPENDOOR (LAMBDA (OPEN ←DOOR)
         (PROG (DECLARE P1)
              (ATTEMPT (EXISTS (OPEN $DOOR) MODELVALUE FALSE)
                                      THEN
                                      (RETURN (OPENDOOR $DOOR)))
              [SETQ ←P1 (GOAL $LOOK (TESTABLE (OPEN $DOOR]
              (ASSERT (OPEN $DOOR))
              (RETURN (TUPLE $$P1 (IF (NOT (OPEN $DOOR))
                                        THEN
                                        (OPENDOOR $DOOR]   .
```

B. TESTABLE

When a plan is constructed to check the door, the predicate (OPEN DOOR) is either asserted or denied in the model, corresponding, for example, to the two cases of conditional planning. Thus the planner assumes that the door has been checked, the predicate has been given a value, and the robot has begun to proceed according to its plan using the value. If such a plan step has been constructed, and the context in which it was constructed is still valid, the statement

(GOAL $LOOK (TESTABLE (OPEN DOOR)))

will always invoke the procedure CHECKMODEL first. This is because (TESTABLE ←X) will not be found in the model and CHECKMODEL is the first program of the GOALCLASS $LOOK.

```
[RPAQQ CHECKMODEL (LAMBDA (TESTABLE ←X)
         (PROG (DECLARE)
               (ATTEMPT (EXISTS $X)
                        THEN
                        (RETURN (TUPLE)))
               (ATTEMPT (EXISTS $X MODELVALUE FALSE)
                        THEN
                        (RETURN (TUPLE)))
               (FAIL]   .
```

CHECKMODEL simply interrogates the model to see if the predicate, in this case (OPEN DOOR), is known either to be TRUE or FALSE. If it is known, CHECKMODEL returns the empty tuple, and no precondition planning steps are generated.

If (CHECKDOOR DOOR) has not been appropriately built into the plan, (OPEN DOOR) will not be known, CHECKMODEL will fail, and the goal interpreter will continue down the programs of GOALCLASS $LOOK to attempt to satisfy the goal. It will find the program CHECKDOOR, which is written in the same framework as OPENDOOR.

```
[RPAQQ CHECKDOOR (LAMBDA (TESTABLE (OPEN ←DOOR))
        (PROG (DECLARE P1)
        (SETQ ←P1 (GOAL $DO (NEXTTO ROBOT $DOOR)))
        (ASSERT (TESTABLE (OPEN $DOOR)))
        (RETURN (TUPLE $$P1 (CHECKDOOR $DOOR]   .
```

To execute the operator CHECKDOOR, it must get the robot to the door. Instead of directly testing the preconditions, it builds a plan that guarantees the robot will be at the door. The plan may be null if the robot is already there, or it may involve many steps if the robot is in another room. In either event, CHECKDOOR asserts that OPENDOOR is now testable, and returns the plan that makes it testable.

C. <u>(GOAL $DO (OPEN DOOR))</u>

The typed in goal statement (1) is executed by the interpreter (2), and the variable GOALCLASS is automatically set by the interpreter (3). The predicate (OPEN DOOR) is not in the model with value TRUE, so the EXISTS portion of the GOAL fails (4), and the operator OPENDOOR is called (5) to work on the goal. The argument is bound (6), and the goal to plan for the preconditions (7,8) is established. (TESTABLE (OPEN DOOR)) is not true in the model, so the EXISTS portion of the GOAL fails (9) and the program CHECKMODEL is called (10). Its argument is bound (11), and it discovers (12,13,14,15) that the predicate is not true in the model, so it also fails (16,17).

The goal interpreter continues down the GOALCLASS $LOOK to CHECKDOOR (18). The argument is bound (19), and the goal is established to get the robot to the DOOR (20,21). The EXISTS portion of the GOAL fails (22), so the GOTO operator is invoked (23 through 29). (See the TURNONLIGHT example in Chapter 2.) CHECKDOOR saves the plan from the GOTO operator in P1 (30) and returns the composite plan to OPENDOOR (31). OPENDOOR saves the composite plan in its local variable P1 (32). OPENDOOR asserts that the door is open (33) and returns the full plan (34) for the original goal (1).

INFORMATION GATHERING

```
 1.  ←! (GOAL $DO (OPEN DOOR]
 2.  (GOAL $DO (OPEN DOOR))
 3.  SETVALUE GOALCLASS (TUPLE LOOPPLAN MOVEBOX OPENDOOR GOTO)
 4.  FAILURE
 5.      LAMBDA OPENDOOR
 6.      SETVALUE DOOR DOOR
 7.      (GOAL $LOOK (TESTABLE (OPEN $DOOR)))
 8.      SETVALUE GOALCLASS (TUPLE CHECKMODEL CHECKDOOR)
 9.      FAILURE
10.          LAMBDA CHECKMODEL
11.          SETVALUE X (OPEN DOOR)
12.          (EXISTS $X)
13.          FAILURE
14.          (EXISTS $X MODELVALUE FALSE)
15.          FAILURE
16.          (FAIL)
17.          FAILURE
18.          LAMBDA CHECKDOOR
19.          SETVALUE DOOR DOOR
20.          (GOAL $DO (NEXTTO (TUPLE ROBOT $DOOR)))
21.          SETVALUE GOALCLASS (TUPLE LOOPPLAN MOVEBOX OPENDOOR GOTO)
22.          FAILURE
23.            LAMBDA GOTO
24.            SETVALUE X DOOR
25.              LAMBDA DELETE
26.              SETVALUE EXP (NEXTTO (TUPLE ROBOT ←Y))
27.              (EXISTS $EXP)
```

```
28.             FAILURE
29.          (ASSERT (NEXTTO (TUPLE ROBOT $X)))
30.       SETVALUE P1 (TUPLE (GOTO DOOR))
31.       (ASSERT (TESTABLE (OPEN $DOOR)))
32.    SETVALUE P1 (TUPLE (GOTO DOOR) (CHECKDOOR DOOR))
33.    ASSERT (OPEN $DOOR))

34. (TUPLE (GOTO DOOR) (CHECKDOOR DOOR) (IF (NOT (OPEN DOOR)) THEN (OPENDOOR
    DOOR)))
    T
    ←
```

IV LOOPS

A. When To versus How To

Our sample program LOOPPLAN demonstrates how a looping program may be validly generated; it does not illustrate the much more difficult problem of discovering the necessity of a loop in the plan. LOOPPLAN is given QA4 programs (or plans) P and Q such that P is a predicate with X free in P, and Q is an action with X free in Q. It is to construct a plan that takes action Q on all objects in the set defined by predicate P. The problems are presented to LOOPPLAN in the form

(FA ←X (IF ←P THEN ←Q)) .

This form is not to be mistaken with the logical universally quantified form. Since the logical form has no interpretation in QA4, and since the semantics are similar, we may use it to represent our goal.

B. LOOPPLAN

```
[RPAQQ LOOPPLAN (LAMBDA (FA ←X (IF ←P THEN ←Q))
     (PROG (DECLARE NC BODY NV1 NV2)
         (EXISTS (GENERABLE $P))
         (SETQ ←NC (CONTEXT PUSH CURRENT))
         (ASSERT $P WRT $NC)
         (SETQ ←BODY (GOAL $GOALCLASS $Q WRT $NC))
         (SETQ (TUPLE ←NV1 ←NV2)
             ($GENVAR (TUPLE)))
         (SETQ ←P (SUBST $P (TUPLE $X $NV1)))
         (SETQ ←BODY (SUBST $BODY (TUPLE $X $NV2)))
         (RETURN (REPEAT (GOAL :$FIND $P)
                   (DO $BODY]
```

LOOPPLAN has three major steps. First it shows that P is GENERABLE; that is, that P can be enumerated when the plan is executed. For our simple planner, we merely assert (GENERABLE P).

Next it derives a new context, assumes P is true, and solves the goal Q. This produces a plan that accomplishes the action Q for the objects in the set defined by P. A careful statement of the problem has assured that the plan is guaranteed to work for any object in the set. Take, for example,

(GOAL $DO (FA BOX (IF (INROOM BOX ROOM) THEN (INROOM BOX LAB)))) .

X is BOX, P is (INROOM BOX ROOM), and Q is (INROOM BOX LAB). While BOX is free in both P and Q, it is a QA4 constant, and plays the role of a general box. It must have the attributes of all boxes, and since it is a constant, will never be instantiated while the planner is constructing a plan to accomplish Q.

When the plan has been accomplished, the constant that is free in both P and Q, and to which the variable X is bound, is replaced by an appropriate variable in both P and Q. These new forms of P and Q are then put together to form the looping plan.

C. <u>(GOAL $DO (FA ...))</u>

The typed in goal (1) is interpreted (2,3) and not found to be true in the model (4). LOOPPLAN is called to solve the goal (5) and its arguments are bound (6,7,8). (INROOM BOX ROOM) is found to be

generable (9) and a new context is generated (10). (INROOM BOX ROOM) is asserted in the new context (11) and (INROOM BOX LAB) is established (12,13) with the same GOALCLASS as the original goal (1).

(INROOM BOX LAB) is not found to be true in the model (14), so MOVEBOX is called (15), and its arguments are bound (16). It is successful (17), and returns a plan which is stored in the variable BODY (18). ←X is substituted for BOX in P (19), and $X is substituted for BOX in Q (20). Finally the looping plan is returned (21).

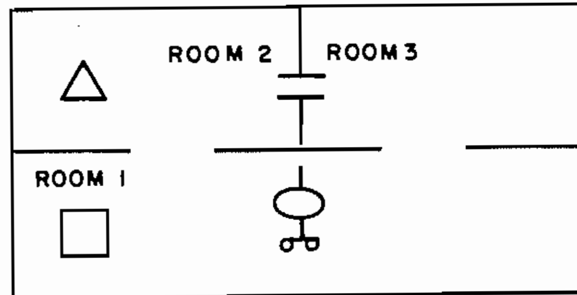
LOOPS

```
1.  ! (GOAL $DO (FA BOX (IF (INROOM BOX ROOM) THEN (INROOM BOX LAB]
2.  (GOAL $DO (FA BOX (IF (INROOM (TUPLE BOX ROOM)) THEN (INROOM
    (TUPLE BOX LAB)))))
3.  SETVALUE GOALCLASS (TUPLE LOOPPLAN MOVEBOX)
4.  FAILURE
5.    LAMBDA LOOPPLAN
6.    SETVALUE Q (INROOM (TUPLE BOX LAB))
7.    SETVALUE P (INROOM (TUPLE BOX ROOM))
8.    SETVALUE X BOX
9.    (EXISTS (GENERABLE $P))
10.   SETVALUE NC (CONTEXT (3 2 1 0) 4 3 1)
11.   (ASSERT $P WRT $NC)
12.   (GOAL $GOALCLASS $Q WRT $NC)
13.   SETVALUE GOALCLASS (TUPLE LOOPPLAN MOVEBOX)
14.   FAILURE
15.     LAMBDA MOVEBOX
16.     SETVALUE BOX BOX
17.     (ASSERT (INROOM (TUPLE $BOX LAB)))
18.   SETVALUE BODY (PUSHBOXTOLAB BOX)
19.   SETVALUE P (INROOM (TUPLE ←X ROOM))
20.   SETVALUE BODY (PUSHBOXTOLAB $X)
21. (REPEAT (GOAL (TUPLE LOCATE CHECKMAP) (INROOM (TUPLE ←X ROOM))) DO
    (PUSHBOXTOLAB $X))
    T
    ←
```

V CONSTRAINTS

A. "Thou shalt not ..." Problems

Suppose our robot is operating in a three-room environment, where one room contains a wedge, another contains a box, and the third is empty.



The task for the robot is to push the box to ROOM1. However, we wish to make the constraint that the box and the wedge may never be in a room simultaneously.

B. Four-Step Operators

Our planner solves the problem by another modification of the handling of operator preconditions. Normally, an operator establishes the preconditions as goals, and then appends its execution operator to the plan for establishing the preconditions. To handle constraints, we will assume the final state, formulate plans necessary to abide by the constraints, then formulate plans to establish the preconditions, and finally return a plan composed of three parts: the constraint satisfaction plans, the precondition plans, and the execution operator.

The PUSHIN operator is supposed to push a box into a room. It normally moves the robot to the box, updates the model, and returns the plan. For the wedge-box example, we insert an initial step. This new operator step tries out the final state and returns a plan, if one is necessary, that resolves conflicts before they arise, such as having the wedge and box in the same room.

```
[RPAQQ PUSHIN (LAMBDA (INROOM ←OBJ ←NEWROOM)
         (PROG (DECLARE OLDROOM PLAN1 PLAN2)
              [SETQ ←PLAN1 ($TRY (' (INROOM $OBJ $NEWROOM]
              (SETQ ←PLAN2 (GOAL $DO (NEXTTO ROBOT $OBJ)))
              ($DELETE (' (INROOM $OBJ ←OLDROOM)))
              (ASSERT (INROOM $OBJ $NEWROOM))
              (RETURN (TUPLE $$PLAN1 $$PLAN2 (PUSHOBJTOROOM $OBJ
                                                        $NEWROOM] .
```

C. Trying Out the Final State

The program that tries out the final state has, as its argument, the pattern that will be asserted by the operator. PUSHIN, for example, calls TRY with the expression (INROOM BOX ROOM2), because that is the predicate it will assert in the model. TRY uses constraint checking programs to examine the potential assertion. When the constraint checking programs discover a violation, they return a pattern expressing a condition, called C1 in the TRY program, that is to be avoided.

TRY then establishes the goal (NOT C1). Operators are called to work on this goal, and the plan they return is appended to the front of the plan TRY is composing. This constraint-checking goal-establishing procedure is repeated until no constraint violations are found. At this time, TRY returns a plan that guarantees the constraints will not be violated.

```
[RPAQQ TRY (LAMBDA ←X
         (PROG (DECLARE C1 PLAN1)
               (SETQQ ←PLAN1 (TUPLE))
           TOP (ATTEMPT (SETQ ←C1 (GOAL $CONSTRAINTS $X))
                        THEN
                        (SETQ ←PLAN2 (GOAL $DO (NOT $C1)))
                        (SETQ ←PLAN1 (APPEND $PLAN2 $PLAN1))
                        (GO TOP)
                        ELSE
                        (RETURN $PLAN1))
               (FAIL]
```

D. Remaining Planner Parts

1. Constraint Checking Programs

For our simple example, there are only two constraint checking programs. If the task is to push the WEDGE to ROOM, the potential assertion of PUSHIN is (INROOM WEDGE ROOM). Thus there is a constraint checking program, BOXCHECK, that applies to expressions of the form

(INROOM WEDGE ←ROOM) and checks for a BOX in ROOM. If one is found, it returns the expression (INROOM BOX $ROOM) to TRY, for TRY must now plan for (NOT (INROOM BOX ROOM)). The BOXCHECK program is:

```
[RPAQQ BOXCHECK (LAMBDA (INROOM WEDGE ←ROOM)
        (EXISTS (INROOM BOX $ROOM]   .
```

The corresponding constraint checking program, that will apply when PUSHIN is moving a BOX is WEDGECHECK:

```
[RPAQQ WEDGECHECK (LAMBDA (INROOM BOX ←ROOM)
        (EXISTS (INROOM WEDGE $ROOM]   .
```

The GOALCLASS CONSTRAINT is:

```
(TUPLE BOXCHECK WEDGECHECK)   .
```

2. PUSHOUT

The PUSHOUT operator will generate a plan for goals of the form (NOT (INROOM ←OBJ ←ROOM)). It operates just like the PUSHIN operator, except that it has the extra step of choosing a room that is connected to $ROOM into which it will push $OBJ. The operator is:

```
[RPAQQ PUSHOUT
      (LAMBDA (NOT (INROOM ←OBJ ←OLDROOM))
        (PROG (DECLARE NEWROOM DOOR PLAN1 PLAN2)
              (EXISTS (CONNECTS ←DOOR (SET $OLDROOM ←NEWROOM)))
              [SETQ ←PLAN1 ($TRY (' (INROOM $OBJ $NEWROOM]
              (SETQ ←PLAN2 (GOAL $DO (NEXTTO ROBOT $OBJ)))
              ($DELETE (' (INROOM $OBJ $OLDROOM)))
              (ASSERT (INROOM $OBJ $NEWROOM))
              (RETURN (TUPLE $$PLAN1 $$PLAN2
                                          (TUPLE (PUSHOBJTOROOM $OBJ $NEWROOM]   .
```

E. (GOAL $DO (INROOM BOX ROOM2))

The typed in goal (1) is interpreted (2,3), and does not exist in the model (4). The first applicable program of the $DO is called (5). This is the operator PUSHIN, and its arguments are immediately bound (6,7). PUSHIN's four phases are the call to TRY (8), planning for the preconditions (63,64), modification of the model (76 through 82), and returning the composite plan (83). Trace steps 63 through 83 do not involve constraints, and proceed as a straightforward, normal operator.

The first phase of PUSHIN, the call to TRY (8), also has four phases:

A--the constraints are checked (11 through 17);

B--a plan is constructed to avoid violation of the constraints (18 through 52);

C--the plan just generated is appended to the plan to avoid all violations (currently empty) and the constraints are checked again (54 through 61); and

D--the complete plan to avoid constraint violations is returned (62).

1. Phase A--Initial Constraint Check

The goal (INROOM BOX ROOM2) is established with GOALCLASS (TUPLE WEDGECHECK BOXCHECK) (11,12). Since (INROOM BOX ROOM2) is not in the model (13), the WEDGECHECK program is applied (14,15). It

notices (INROOM WEDGE ROOM2) (16) is the condition that causes the potential constraint violation, and returns it to TRY, which saves it in variable C1 (17).

2. Phase B--Planning to Avoid the Constraint

TRY now establishes (NOT (INROOM WEDGE ROOM2)) as a goal to be solved (18,19). The goal is not true in the model (20), so the operator PUSHOUT is applied (21,22,23). PUSHOUT has five steps. The first chooses a room into which it will push the WEDGE (24,25,26). The last four phases correspond to the four phases of PUSHIN: it calls TRY (27), plans for the preconditions (35), modifies the model (46 through 51), and returns the plan to TRY (52). The last three steps proceed as a normal operator (35 through 52).

The recursive call to TRY (27), will check if moving WEDGE to ROOM3 violates any constraints (30,31). Neither constraint checking program can find a violation so they both fail (32,33). Thus TRY returns the empty tuple, and it is saved by PUSHOUT in the variable PLAN1 (34). With this, PUSHOUT proceeds through its last three phases, and returns to TRY a plan that pushes the WEDGE out of ROOM2 and into ROOM3.

3. Phase C--Checking for More Constraints

The plan to move the WEDGE is saved in the variable PLAN1 of TRY (53), and TRY loops back to the constraint check. Now that PUSHOUT has been executed, the model reflects the movement of the WEDGE. Even though WEDGECHECK is called (57,58), the original potential assertion (INROOM BOX ROOM2) no longer violates any constraints (59,60,61).

4. Phase D

TRY has now completed its checking procedure, and returns the plan it generated to PUSHIN, which saves it in the variable PLAN1 (62).

CONSTRAINTS

```
1.   ←! (GOAL $DO (INROOM BOX ROOM2]
2.   (GOAL $DO (INROOM (TUPLE BOX ROOM2)))
3.   SETVALUE GOALCLASS (TUPLE PUSHIN PUSHOUT GOTO LOOPPLAN MOVEBOX OPENDOOR)
4.   FAILURE
5.     LAMBDA PUSHIN
6.     SETVALUE NEWROOM ROOM2
7.     SETVALUE OBJ BOX
8.       LAMBDA TRY
9.       SETVALUE X (INROOM (TUPLE BOX ROOM2))
10.      SETVALUE PLAN1 (TUPLE)
11.      (GOAL $CONSTRAINTS $X)
12.      SETVALUE GOALCLASS (TUPLE WEDGECHECK BOXCHECK)
13.      FAILURE
14.        LAMBDA WEDGECHECK
15.        SETVALUE ROOM2
16.        (EXISTS (INROOM (TUPLE WEDGE $ROOM)))
17.      SETVALUE C1 (INROOM (TUPLE WEDGE ROOM2))
18.      (GOAL $DO (NOT $C1))
19.      SETVALUE GOALCLASS (TUPLE PUSHIN PUSHOUT GOTO LOOPPLAN MOVEBOX
         OPENDOOR)
20.      FAILURE
21.        LAMBDA PUSHOUT
22.        SETVALUE OLDROOM ROOM2
23.        SETVALUE OBJ WEDGE
24.        (EXISTS (CONNECTS (TUPLE ←DOOR (SET $OLDROOM ←NEWROOM))))
25.        SETVALUE DOOR DOOR23
26.        SETVALUE NEWROOM ROOM3
27.          LAMBDA TRY
```

```
28.        SETVALUE X (INROOM (TUPLE WEDGE ROOM3))
29.        SETVALUE PLAN1 (TUPLE)
30.        (GOAL $CONSTRAINTS $X)
31.        SETVALUE GOALCLASS (TUPLE WEDGECHECK BOXCHECK)
32.        FAILURE
33.        FAILURE
34.      SETVALUE PLAN1 (TUPLE)
35.      (GOAL $DO (NEXTTO (TUPLE ROBOT $OBJ)))
36.      SETVALUE GOALCLASS (TUPLE PUSHIN PUSHOUT GOTO LOOPPLAN MOVEBOX
         OPENDOOR)
37.      FAILURE
38.        LAMBDA GOTO
39.        SETVALUE X WEDGE
40.          LAMBDA DELETE
41.          SETVALUE EXP (NEXTTO (TUPLE ROBOT ←Y))
42.          (EXISTS $EXP)
43.          FAILURE
44.        (ASSERT (NEXTTO (TUPLE ROBOT $X)))
45.      SETVALUE PLAN2 (TUPLE (GOTO WEDGE))
46.        LAMBDA DELETE
47.        SETVALUE EXP (INROOM (TUPLE WEDGE ROOM2))
48.        (EXISTS $EXP)
49.        SETVALUE X (INROOM (TUPLE WEDGE ROOM2))
50.        (DENY $X)
51.      (ASSERT (INROOM (TUPLE $OBJ $NEWROOM)))
52.    SETVALUE PLAN2 (TUPLE (GOTO WEDGE) (PUSHOBJTOROOM (TUPLE WEDGE
       ROOM3)))
53.    SETVALUE PLAN1 (TUPLE (GOTO WEDGE) (PUSHOBJTOROOM (TUPLE WEDGE
       ROOM3)))
```

```
54.    (GOAL RCONSTRAINTS $X)
55.    SETVALUE GOALCLASS (TUPLE WEDGECHECK BOXCHECK)
56.    FAILURE
57.      LAMBDA WEDGECHECK
58.      SETVALUE ROOM ROOM2
59.      (EXISTS (INROOM (TUPLE WEDGE $ROOM)))
60.      FAILURE
61.    FAILURE
62.  SETVALUE PLAN1 (TUPLE (GOTO WEDGE) (PUSHOBJTOROOM (TUPLE WEDGE ROOM3)))

63.  (GOAL $DO (NEXTTO (TUPLE ROBOT $OBJ)))
64.  SETVALUE GOALCLASS (TUPLE PUSHIN PUSHOUT GOTO LOOPPLAN MOVEBOX
     OPENDOOR)
65.  FAILURE
66.    LAMBDA GOTO
67.    SETVALUE X BOX
68.      LAMBDA DELETE
69.      SETVALUE EXP (NEXTTO (TUPLE ROBOT ←Y))
70.      (EXISTS $EXP)
71.      SETVALUE Y WEDGE
72.      SETVALUE X (NEXTTO (TUPLE ROBOT WEDGE))
73.      (DENY $X)
74.    (ASSERT (NEXTTO (TUPLE ROBOT $X)))
75.  SETVALUE PLAN2 (TUPLE (GOTO BOX))
76.    LAMBDA DELETE
77.    SETVALUE EXP (INROOM (TUPLE BOX ←OLDROOM))
78.    (EXISTS $EXP)
79.    SETVALUE OLDROOM ROOM1
```

```
80.      SETVALUE X (INROOM (TUPLE BOX ROOM1))
81.      (DENY $X)
82.   (ASSERT (INROOM (TUPLE $OBJ $NEWROOM)))
83.   (TUPLE (GOTO WEDGE) (PUSHOBJTOROOM (TUPLE WEDGE ROOM3)) (GOTO BOX)
      (PUSHOBJTOROOM (TUPLE BOX ROOM2)))
      T
      ←
```

VI COORDINATED PLANS

A. A Shopping Problem

As a final problem, suppose our robot wishes to have some yogurt and mail a letter. The GROCERY operator's structure resembles our other operators. It works on the goal (HAS ←X).

```
[RPAQQ GROCERY (LAMBDA (HAS ←X)
         (PROG (DECLARE)
               (GOAL $PRECONDITION (HAS MONEY))
               [WAIT (' (TUPLE (GOTO GROCERY)
                               (BUY $X]
               (SETQ ←YOUAREHERE GROCERY)
               (ASSERT (HAS $X))
               (WAIT DONE]   .
```

First it makes sure the robot has money, then constructs the plan to go to the grocery and buy yogurt, and finally modifies the model to reflect these last plan steps--namely (ASSERT (HAS YOGURT)) and (SETQ ←YOUAREHERE GROCERY).

The operators in this example are invoked as procedures rather than ordinary function calls. As we will see later, this permits the planning of coordinated actions. It requires, however, that the planning steps be sent to the calling programs with WAIT statements rather than the RETURN statements used in the previous examples. The use of the WAIT statements greatly enhances the power of the operators.

When executed, they send their planning steps as a message to the process that created them. When they are resumed ("called" for a process) they are given a message and continue operation at the statement following the WAIT. This means they can intersperse preconditions and planning steps. A MAILLETTER operator could, for example, have the steps

(1) preconditions	(HAS ENVELOPE)
(2) plan step	(PUT LETTER IN ENVELOPE)
(3) precondition	(HAS STAMP)
(4) plan step	(PUT STAMP ON ENVELOPE)
(5) precondition	(BE AT A MAILBOX)
(6) plan step	(MAIL LETTER) .

The structure of our MAILLETTER operator, however, resembles the GROCERY operator. First it plans for the precondition (AND (HAS STAMP) (HAS ENVELOPE)), and then constructs the plan step (WHEN EXP (SEE MAILBOX) THEN (MAIL LETTER)).

```
[RPAQQ MAILLETTER (LAMBDA (SENT LETTER)
        (PROG (DECLARE)
              (GOAL $PRECONDITION (AND (HAS STAMP)
                                       (HAS ENVELOPE)))
              [WAIT (' (TUPLE (WHEN EXP (SEE MAILBOX)
                                   THEN
                                   (MAILLETTER]
              (WAIT DONE]   .
```

It does not update the model. The WHEN statement it adds to the plan establishes a demon during plan execution. The demon will watch for a MAILBOX, and mail a letter when it notices one. We could, in a more sophisticated planner, both build a WHEN statement into the plan and establish a demon in the planner. Then as the planner progressed, it would update the model to indicate the letter was sent when, during planning, the robot passed a mailbox in the model.

We have chosen this structure rather than the multiple step structure mentioned above because the preconditions are unordered and we usually mail letters only when we see a mailbox, not as the next step after putting the stamp on the envelope. Completely ordered preconditions may, at times, lead to a silly plan. Suppose we did not have either a stamp or an envelope, and our street map was:

```
(RPAQQ STREETMAP (TUPLE GROCERY STATIONERY BANK POSTOFFICE))
```

(The mailbox is between GROCERY and the STATIONERY, and the POSTOFFICE is a stamp machine, so letters must be mailed at the mailbox.) If we force an order on the operations of buying the envelope, buying the stamp, and mailing the letter we produce a poor plan.

Instead of forcing an order on the preconditions, the goal of (AND (HAS STAMP) (HAS ENVELOPE)) will generate a plan that assures we have both items, but it may gather them in either order. The ability to deal with unordered preconditions will permit our robot to better plan for the original problem: (AND (HAS YOGURT) (SENT LETTER)).

The POSTOFFICE and STATIONERY operators are similar to the GROCERY operator:

```
[PRAQQ POSTOFFICE (LAMBDA (HAS STAMP)
        (PROG (DECLARE)
              (GOAL $PRECONDITION (HAS MONEY))
              [WAIT ←YOUAREHERE POSTOFFICE)
              (ASSERT (HAS STAMP))
              (WAIT DONE]
[RPAQQ STATIONERY (LAMBDA (HAS ENVELOPE)
        (PROG (DECLARE)
              (GOAL $PRECONDITION (HAS MONEY))
              [WAIT (' (TUPLE (GOTO STATIONERY)
                              (BUY ENVELOPE]
              (SETQ ←YOUAREHERE STATIONERY)
              (ASSERT (HAS ENVELOPE))
              (WAIT DONE]   .
```

Notice that they both have the same precondition as GROCERY: that the robot must have money before he goes shopping.

The BANK operator lacks preconditions, and has only one main step--constructing the single planning step (TUPLE (GOTO BANK) (CASH CHECK)).

```
[RPAQQ BANK (LAMBDA (HAS MONEY)
          (PROG (DECLARE)
                [WAIT (' (TUPLE (GOTO BANK)
                                (CASH CHECK]
                (SETQ ←YOUAREHERE BANK)
                (ASSERT (HAS MONEY))
                (WAIT DONE]
```

B. The Shopping List

The planner must somehow coordinate preconditions so that the shopping trip is executed in a reasonable order. In this case, the initial goal (AND (HAS YOGURT) (SENT LETTER)) uses the operators GROCERY and MAILLETTER. GROCERY establishes the precondition (HAS MONEY). BANK satisfies this condition, thus the first step of GROCERY will be (TUPLE (GOTO BANK) (CASH CHECK)). MAILLETTER establishes the precondition (AND (HAS STAMP) (HAS ENVELOPE)). POSTOFFICE works on (HAS STAMP) and STATIONERY works on (HAS ENVELOPE). Both POSTOFFICE and STATIONERY also have the precondition (HAS MONEY). For coordinated planning we must notice this and first go to the bank, then go to each store, making a single trip through town, and begin to watch for a mailbox as soon as we have both a stamp and an envelope.

Our robot plans from a shopping list. The shopping list contains all the possible next steps for the final plan. The planner chooses the next step based on its location in town and the direction it is

going, and adds that step to the general plan. Associated with the step just added to the plan was an operator, and the planner then adds to the shopping list the next step from that operator. When an operator is DONE, it is removed from the shopping list.

If an operator wishes to suspend itself from consideration, until a precondition is true, it is included on the shopping list in conditional form. On each iteration through the list, the condition is checked; if it is true, the operator is resumed, and its next step is then included on the list.

Let us first step through the initial building of the shopping list in an intuitive way, bypassing the details of the processes and their methods of interaction. The goal (HAS YOGURT) creates a process for GROCERY, say G, which has two effects. First it creates a process for BANK, say GB, which adds (TUPLE (GOTO BANK) (CASH CHECK)) to the shopping list. Then GROCERY adds itself to the shopping list as a conditional element. Thus (HAS YOGURT) initially makes the shopping list into

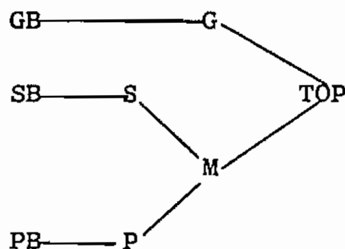
```
(TUPLE     (TUPLE BG (TUPLE (GOTO BANK) (CASH CHECK)))
           (TUPLE  G (TUPLE CONDITIONAL (HAS MONEY]   .
```

The goal (SENT LETTER) creates a process for the MAILLETTER operator, say, M. This operator has an AND precondition similar to the initial AND goal for the problem. So M creates a process for STATIONERY, say S. S immediately creates another BANK process, say SB. The AND precondition

for MAILLETTER then creates a process for POSTOFFICE, say P, that creates yet another process for BANK, say PB. Finally, MAILLETTER adds itself to the shopping list as a conditional element. At the end of the first phase of planning we have a shopping list of seven items:

```
(TUPLE      (TUPLE GB (TUPLE (GOTO BANK) (CASH CHECK)))
            (TUPLE G  (TUPLE CONDITIONAL (HAS MONEY)))
            (TUPLE SB (TUPLE (GOTO BANK) (CASH CHECK)))
            (TUPLE S  (TUPLE CONDITIONAL (HAS MONEY)))
            (TUPLE PB (TUPLE (GOTO BANK) (CASH CHECK)))
            (TUPLE P  (TUPLE CONDITIONAL (HAS MONEY)))
            (TUPLE M  (TUPLE CONDITIONAL
                        (AND (HAS STAMP) (HAS ENVELOPE]   .
```

Internally, the planner has built a structure of processes that resembles, in many ways, a critical path or PERT chart for accomplishing the tasks. Our chart-builting process has not identified equal nodes; that will be done during the iterative cycles over the shopping list. The chart for this example is:



C. Sorting the List

To generate the plan, we enter a second phase. The planner will choose a next step from the shopping list and add it to the plan. This step is replaced by a new next step from the appropriate operator and the whole list is considered again. When an operator is DONE, it is removed from the list. Let us step through the sorting process in the same intuitive way we described its construction.

On the first pass through the initial list of seven items, none of the conditionals are true, so all four conditional items remain on the list. The remaining three items are all the same. Two are removed, and the third is considered the best. The plan step for the best item is appended to the main plan and the process associated with the best item is resumed to get its next step. Since it is one of the BANK items, it returns DONE. When it is resumed, it has the important side effect of asserting (HAS MONEY). Thus, at the completion of the first cycle, the plan is

```
        (TUPLE (TUPLE (GOTO BANK) (CASH CHECK)))
```

and the shopping list is:

```
    (TUPLE        (TUPLE GB DONE)
                  (TUPLE G (TUPLE CONDITIONAL (HAS MONEY)))
                  (TUPLE S (TUPLE CONDITIONAL (HAS MONEY)))
                  (TUPLE P (TUPLE CONDITIONAL (HAS MONEY)))
                  (TUPLE M (TUPLE CONDITIONAL
                                  (AND (HAS STAMP) (HAS ENVELOPE]   .
```

During the second cycle through the list, GB is discarded. This time, however, the condition (HAS MONEY) is true. The process G is resumed and returns (TUPLE (GOTO GROCERY) (BUY YOGURT)) as its next step. Then this next step is compared to other next steps and is either considered BEST or put on the shopping list. In this case, since it is the first step considered during the cycle, it is considered BEST.

(TUPLE S (TUPLE CONDITIONAL (HAS MONEY))) is considered on the second step of the cycle through the shopping list. Since (HAS MONEY) is true, S is resumed and returns, as a next step, (TUPLE (GOTO STATIONERY) (BUY ENVELOPE)). According to the street map, this is a better next step because the STATIONERY is on the way to the GROCERY. Thus it is made BEST and (TUPLE G (TUPLE (GOTO GROCERY) (BUY YOGURT))) is added to the new shopping list.

When (TUPLE P (TUPLE CONDITIONAL (HAS MONEY))) is considered, a similar event occurs. But since going to the POSTOFFICE would require backtracking in the plan, it is not better than going to the STATIONERY. Thus (TUPLE P (TUPLE (GOTO POSTOFFICE) (BUY STAMP))) is added to the new shopping list. Finally, when (TUPLE M (TUPLE CONDITIONAL (AND (HAS STAMP) (HAS ENVELOPE)))) is considered, the condition is not true, so it remains on the shopping list. Having considered all four items on this second cycle, the planner appends (TUPLE (GOTO STATIONERY) (BUY ENVELOPE)) to the plan, resumes S, and includes (TUPLE S DONE) on the shopping list.

On the third cycle, GROCERY is BEST, and the shopping list becomes:

```
(TUPLE        (TUPLE G DONE)
              (TUPLE P (TUPLE CONDITIONAL (HAS MONEY)))
              (TUPLE M (TUPLE CONDITIONAL
                 (AND (HAS STAMP) (HAS ENVELOPE]   .
```

On the fourth cycle, the condition for M does not yet hold, so P is BEST.

At the beginning of the fifth cycle, the plan is

```
(TUPLE        (TUPLE (GOTO BANK) (CASH CHECK))
              (TUPLE (GOTO GROCERY) (BUY YOGURT))
              (TUPLE (GOTO STATIONERY) (BUY ENVELOPE))
              (TUPLE (GOTO POSTOFFICE) (BUY STAMP]   .
```

Since S and P both have been resumed (HAS STAMP) and (HAS ENVELOPE) have both been asserted. Thus the condition for M will hold. When it is resumed, its next step is included in the plan. On the sixth cycle, the shopping list is reduced to the empty tuple, and the plan is completed.

D. Process Control Programs

1. Initial Construction

The program SHOP sets up the initial model for our problem.

```
[RPAQQ SHOP (LAMBDA ←X
        (PROG (DECLARE SHOPPINGLIST PLAN1)
              (SETQ ←COMING FALSE)
              (SETQ ←SHOPPINGLIST (TUPLE))
              (SETQ ←PLAN1 (TUPLE))
              (DENY (HAS MONEY))
              (DENY (HAS YOGURT))
              (DENY (HAS STAMP))
              (DENY (HAS ENVELOPE))
              (PUT (SENT LETTER)
                   WHERE $MAILLETTER)
              (PUT (HAS STAMP)
                   WHERE $POSTOFFICE)
              (PUT (HAS YOGURT)
                   WHERE $GROCERY)
              (PUT (HAS ENVELOPE)
                   WHERE $STATIONERY)
              (PUT (HAS MONEY)
                   WHERE $BANK)
              (GOAL (TUPLE MAPPLAN ONEPLAN)
                    $X)
              (RETURN (= ($SORTPLAN (TUPLE]
```

Its final two statements, GOAL and RETURN, initiate the two main phases of the solution.

On the top level, the programs ONEPLAN and MAPPLAN both use ADDTOLIST to generate the initial shopping list.

```
(RPAQQ ONEPLAN [LAMBDA ←GOAL
        ($ADDTOLIST $GOAL])
(RPAQQ MAPPLAN [LAMBDA (AND ←←GOALSET)
        (MAPC $GOALSET $ADDTOLIST])
[RPAQQ ADDTOLIST (LAMBDA ←GOAL1
        (PROG (DECLARE TASK1)
              (SETQ ←TASK1 (INCARNATE (GET $GOAL1 WHERE)))
              [SETQ ←SHOPPINGLIST (TUPLE $$SHOPPINGLIST
                                            (TUPLE $TASK1 (RESUME
                                                       $TASK1 $GOAL1]
              (RETURN ADDTOLIST]   .
```

Suppose our initial goal were only (HAS YOGURT). Then GOAL statement of SHOP would call ONEPLAN, which would call ADDTOLIST with (HAS YOGURT) as the argument GOAL1. The GET statement of ADDTOLIST finds the program GROCERY. The INCARNATE statement creates a process from this program, called G in the discussion above, and TASK1 is set to this process. The second SETQ statement of ADDTOLIST performs two steps. First it resumes the process. The value of the RESUME will be the first step for the operator GROCERY. In this case the value of the RESUME will be (TUPLE CONDITIONAL (HAS MONEY)). Then, the step is coupled with TASK1 into a two-tuple and added to SHOPPINGLIST.

When the process G was initially resumed, the interpreter bound the message of the RESUME statement, $GOAL1 or (HAS YOGURT), to the variable X of GROCERY, and began the interpretation of GROCERY. The first statement, (GOAL $PRECONDITION (HAS MONEY)) calls the program SIMPLECONDITION with (HAS MONEY) as the argument.

```
[RPAQQ SIMPLECONDITION (LAMBDA ←GOAL
         (PROG (DECLARE)
               ($ADDTOLIST $GOAL)
               ($WAITGOAL $GOAL)
               (RETURN SIMPLECONDITION]   .
```

SIMPLECONDITION immediately calls ADDTOLIST with (HAS MONEY) as the goal. When ADDTOLIST returns, SIMPLECONDITION then calls WAITGOAL.

```
[RPAQQ WAITGOAL (LAMBDA ←CONDITION
         (PROG (DECLARE NEXTSTEP)
               (SETQ ←NEXTSTEP (TUPLE CONDITIONAL $CONDITION))
          LOOP (SETQ ←NEXTSTEP (WAIT $NEXTSTEP))
               (IF (NOTEQUAL $NEXTSTEP ALL-DONE)
                   THEN
                   (GO LOOP))
               (RETURN WAITGOAL]   .
```

This program sets NEXTSTEP to (TUPLE CONDITIONAL (HAS MONEY)) and executes a WAIT with NEXTSTEP as the message. A WAIT resumes the mother process, so at this point the RESUME of the initial call to

ADDTOLIST from ONEPLAN is evaluated, and the top level process continues by adding the conditional to SHOPPINGLIST.

The call to ADDTOLIST from SIMPLECONDITION has the side effect of creating a process from (HAS MONEY), called GB above, and resuming it. By the time the RESUME in the initial call to ADDTOLIST is evaluated, two processes have been created.

When a process is resumed by any other process, it begins execution at the point it executed its last WAIT or RESUME. The process B is waiting for a message to become the value of the first WAIT statement in the function BANK. When ADDTOLIST at the top level returns, ONEPLAN has satisfied the GOAL, and SHOP is about to execute the RETURN statement. The process structure is:

STATE	PROCESS
at RETURN	top level
in WAITGOAL loop	G
in BANK	GB

WAITGOAL will play an important role in maintaining conditional processes until their preconditions are true. When the top level process resumes GB, the WAIT in GB will cause G to become activated. Since it is not ALL-DONE, the message will be immediately relayed to the mother of G--the top level process. However, when the top-level

process is ready to activate G directly, it resumes with the message ALL-DONE. This happens during the sorting phase when a condition is true and the next step for a conditional process needs to be considered. WAITGOAL then recognizes ALL-DONE as a message that the precondition is satisfied and returns to SIMPLECONDITION. SIMPLECONDITION returns to GROCERY, which sends a message back to the top-level with a WAIT.

The addition of AND goals to this process structure is trivial. We merely apply ADDTOLIST to each member of the AND set. The GOALCLASS for SHOP on the top level is (TUPLE ONEPLAN MAPPLAN). MAPPLAN is a MAPC extension of ONEPLAN. Within each process, the GOALCLASS for $PRECONDITION is (TUPLE ANDCONDITION SIMPLECONDITION). ANDCONDITION is a MAPC extension of SIMPLECONDITION.

```
[RPAQQ ANDCONDITION (LAMBDA (AND ←GOALSET)
         (PROG (DECLARE)
               (MAPC $GOALSET $ADDTOLIST)
               ($WAITGOAL (' (AND $$GOALSET)))
               (RETURN ANDCONDITION]
```

2. Process Interaction

The program SORTPLAN, executed by the top level process, uses the program MAKEPLAN1 to add the best next step of the plan and create a new shopping list. It simply calls MAKEPLAN1 and checks if SHOPPINGLIST is empty. If not, it repeats the call to MAKEPLAN1.

```
[RPAQQ SORTPLAN (LAMBDA (TUPLE)
         (PROG (DECLARE)
           SORT (SETQ ←PLAN1 ($MAKEPLAN1 $PLAN1))
                (IF (EQUAL $SHOPPINGLIST (TUPLE))
                    THEN
                    (RETURN $PLAN1)
                    ELSE
                    (GO SORT]   .
```

MAKEPLAN1 performs an iteration through the elements of SHOPPINGLIST and chooses the best one.

```
[RPAQQ MAKEPLAN1 (LAMBDA ←BIGPLAN
         (PROG (DECLARE BESTPLAN BESTPROCESS BESTPLACE NEWLIST BEST
                        NEWPLAN)
               (SETQ ←NEWLIST (TUPLE))
               (SETQ ←BEST FALSE)
               (SETQ ←BESTPLACE MIT)
               (SETQ ←BESTPLAN (TUPLE))
               (MAPC $SHOPPINGLIST $FINDNEXTPLACE)
               (IF (EQUAL $BESTPLACE MIT)
                   THEN
                   (SETQ ←NEWLIST (TUPLE))
                   (SETQ ←COMING (NOT $COMING))
                   (MAPC $SHOPPINGLIST $FINDNEXTPLACE))
               (SETQ ←SHOPPINGLIST $NEWLIST)
               (IF $BESTPROCESS THEN (SETQ ←NEWPLAN (RESUME
                                         $BESTPROCESS TRUE))
                   (SETQ ←SHOPPINGLIST (TUPLE (TUPLE $BESTPROCESS
                                                      $NEWPLAN)
                                               $$SHOPPINGLIST)))
               (RETURN (TUPLE $$BIGPLAN $$BESTPLAN]   .
```

First it initializes BEST to be the worst possible place. It then uses a MAPC to apply FINDNEXTPLACE to each member of SHOPPINGLIST. If no BESTPLACE was found, it reverses the direction of the robot from COMING to (NOT COMING), or going, and tries again to find a BESTPLACE. FINDNEXTPLACE creates NEWLIST, a new shopping list. MAKEPLAN1 sets SHOPPING LIST to NEWLIST. Then, if a BESTPROCESS was found, it resumes it in order to get its next plan step. This NEWPLAN together with BESTPROCESS is then added to SHOPPINGLIST and MAKEPLAN1 returns the new plan.

FINDNEXTPLACE analyzes each proposed step.

```
[RPAQQ FINDNEXTPLACE (LAMBDA (PAND ←P (TUPLE ←PROCESS ←PLAN))
        (PROG (DECLARE CONDITION)
              (IF (OR (EQUAL $PLAN DONE)
                      (EQUAL $PLAN $BESTPLAN))
                  THEN
                  (RETURN FINDNEXTPLACE))
              (ATTEMPT (SETQ (TUPLE CONDITIONAL ←CONDITION)
                             $PLAN)
                  THEN
                  (IF (GOAL $CHECK $CONDITION)
                      THEN
                      (SETQ ←PLAN (RESUME $PROCESS ALL-DONE))
                      (SETQ ←P (TUPLE $PROCESS $PLAN))
                      ($COMPAREPLAN $PLAN)
                      ELSE
                      (SETQ ←NEWLIST (TUPLE $P $$NEWLIST)))
                  ELSE
                  ($COMPAREPLAN $PLAN))
              (RETURN FINDNEXTPLACE]   .
```

If the plan is DONE it takes no action, and neither adds anything to NEWLIST, the new SHOPPINGLIST, nor compares PLAN to BESTPLAN. If PLAN is EQUAL to BESTPLAN it also does nothing. This is where the three BANK steps were recognized as the same.

If PLAN is a conditional, it checks the condition using GOALCLASS $CHECK. If the condition is not true, it adds the condition to NEWLIST. If it was true, it resumes the PROCESS with ALL-DONE. This finds the next step for PROCESS. Then the program COMPAREPLAN is used to compare this next step to the BEST so far. If the PLAN was not a conditional, the ELSE clause of the ATTEMPT statement calls COMPAREPLAN directly. COMPAREPLAN first checks to see if the plan's first step is a GOTO. If it is, is uses BETTER to compare the PLACE to BESTPLACE. If the first step is not a GOTO, then COMPAREPLAN assumes it can be executed ANYWHERE, and thus makes it the BEST. If COMPAREPLAN updates BEST, it also updates NEWLIST by adding the old BEST to it.

```
[RPAQQ COMPAREPLAN (LAMBDA (TUPLE ←STEP1 ←←OTHERSTEP)
         (PROG (DECLARE PLACE ISGO ILIKE)
               (SETQ ←PLACE ANYWHERE)
               (SETQ ←ISGO FALSE)
               (SETQ ←ILIKE FALSE)
               (SETQ ←ISGO (ATTEMPT (SETQ (GOTO ←PLACE)
                                       $STEP1)))
               (IF $ISGO THEN (SETQ ←ILIKE ($BETTER $PLACE)))
               (IF (OR (NOT $ISGO)
                       $ILIKE)
                   THEN
                   (IF $BEST THEN (SETQ ←NEWLIST (TUPLE $BEST
                                                        $$NEWLIST)))
                   (SETQ ←BEST $P)
                   (SETQ ←BESTPROCESS $PROCESS)
                   (SETQ ←BESTPLAN $PLAN)
                   (SETQ ←BESTPLACE $PLACE)
                   ELSE
                   (SETQ ←NEWLIST (TUPLE $P $$NEWLIST]   .
```

For this simple shopping problem, STREETMAP is a tuple. In this way BETTER can use a single pattern match with fragment variables to compare PLACE and BESTPLACE.

```
[RPAQQ BETTER
       (LAMBDA ←PLACE
         (PROG (DECLARE X Y Z)
               (IF (EQUAL $BESTPLACE MIT)
                   THEN
                   (RETURN TRUE))
               (IF (EQUAL $BESTPLACE ANYWHERE)
                   THEN
                   (RETURN FALSE))
               (IF (NOT $COMING)
                   THEN
                   (ATTEMPT (SETQ (TUPLE ←←X $YOUAREHERE ←←Y $PLACE
                                          ←←Z $BESTPLACE ←←W)
                            $STREETMAP]    .
```

The GOALCLASS $CHECK checks the MODELVALUE of each item of the condition.

```
(RPAQQ CHECK (TUPLE ANDCHECK SIMPLECHECK))
[RPAQQ ANDCHECK (LAMBDA (AND ←←CONDITIONS)
         (PROG (DECLARE X)
               (SETQ ←X TRUE)
               (MAPC $CONDITIONS $CHECKONECONDITION)
               (RETURN $X]
[RPAQQ CHECKONECONDITION (LAMDBA ←Y
         (SETQ ←X (AND (GET $Y MODELVALUE)
                       $X]
(RPAQQ SIMPLECHECK [LAMBDA ←CONDITION
         (GET $CONDITION MODELVALUE])    .
```

Chapter Four

THE LANGUAGE

I RUNNING THE SYSTEM

A. Warnings

This chapter may be read without an intimate knowledge of TENEX or BBN LISP (Teitelman, 1972). If you wish to run QA4 however, you will find it necessary to READ YOUR MANUAL.

This note contains details on all the system features that work and, additionally, those that are planned for in the future. Some section titles are marked with an asterisk (*) to indicate that these features do not yet work. They may be coded and not debugged, so be careful.

B. Loading the System

To use the QA4 system, one must first LOGIN to TENEX and enter BBN LISP. The most recent version of the QA4 system is kept as a SYSOUT file named QA4.SYS;1 in Rulifson's directory. Thus the LISP command

```
←SYSIN(<RULIFSON>QA4.SYS;1)
```

will load the system. The response should be HELLO. You have then loaded QA4 and are talking to the top level of LISP.

C. Talking to QA4

All communication to QA4 is done through LISPX. The breakout character is !. For example, the command

```
←! (SETQ ←X 3)
```

would set the top-level QA4 value of the variable X to 3. Since LISPX has had a chance to see the input line, all its commands such as FIX and REDO will work properly. The QA4 user language is slightly different from the QA4 internal form for expressions, although typing the internal form will always work. Before QA4 is actually presented an input line, an automatic editing occurs. This initial editing pass will supply many of the key words used in the internal format. For example, the internal form of the above line is

(STATEMENT SETQ (BV X PREFIX ←) 3)

meaning X is a bound variable with prefix ←. The differences in the two formats are discussed throughout this report.

D. Establishing a Program

All QA4 programs must have a name. The user form of the program is stored as the LISP value of the name, and the internal form of the program is stored as the QA4 value of the name. Only the QA4 value is necessary to execute the program. Thus the command

← ! (SETQQ ←FOO (LAMBDA ←X (PLUS ←X 1)))

will set the QA4 variable FOO to an expression that is a QA4 program. The function EDITQ will edit QA4 programs. It first checks for a LISP value for its argument, and if it cannot find one it creates one from the QA4 value. It then calls the LISP editor to permit the user to edit the function. When the editor exits with an OK or SAVE, EDITQ saves the LISP value and generates a new QA4 value. The next time the

function is edited, the edit history list is restored so that old commands may be UNDONE just as with LISP functions. EDITQ is called in the same manner as EDITF.

E. Commands to Establish Models

Usually a file contains commands to the QA4 system as well as functions. These commands are expressions to be evaluated at the top level and establish the model and inform the system about the use of programs to work on goals and establish demons. To save a list of commands in a file, a user first puts the list of commands under the property HISTORY on a LISP atom. To save these properties, add (PROP HISTORY n1 n2 ...) to fileVARS. Each ni is an atom whose LISP value is a list of commands. The format of the list of commands is a list of lists. Each sublist has ! as the first element and the command as the second element. For example,

```
(   ((! (ASSERT (P A)))
    ((! (DENY (P B))))) .
```

Appendix I contains a detailed example of a file with many functions and commands.

II PRIMITIVE EXPRESSIONS

A. Types and Formats

The entities of the QA4 language are expressions, and every expression has a "syntactic type." This feature of an expression indicates the way the expression is to be interpreted. The syntactic types are:

INDENT, SET, BAG, NUMBER, CONTEXT, PROCESS, SEMAPHORE, LAMBDA, FA, EX, APPL, STATEMENT.

This notion of type plays a different role in the interpretation of the language than the more common "data type." In QA4 a tuple and a set both have data type list, but have different syntactic type, and thus have different meaning during evaluation.

There are two formats for QA4 expressions: user, and internal. The user format is a condensed or shorthand version of the internal. As we discuss each of the types and their meaning in the rest of this note, we will point out the differences in these two formats. Any time expressions are printed or edited within the system, the expressions are first converted from internal to user format. The internal format that is used by the interpreter is the key to many of the more complicated operations the interpreter performs on expressions.

Each expression has a standard or canonical internal form. Any time an expression is constructed within the system it is converted to this canonical form. Expressions that are syntactically equivalent are

always converted into the same expression. For example, (SET 2 1) and (SET 1 2) are the same set and are represented within the system by the same expression. Appendix II contains a complete description of this process. This internal format also permits the expressions to have properties. In fact, the expression is a list of properties and the syntactic component identifies the expression. Expressions that are the same up to change of bound variables and set and bag permutations are identified. For example, using mathematical notation rather than QA4 notation, if the expression

$$(\lambda\ (X\ Y)\ (X + Y) * (Y + 1))$$

existed in the system and a program constructed

$$(\lambda\ (U\ V)\ (U + 1) * (V + U)) \quad ,$$

they would be recognized as syntactically equivalent, and only the first would remain in the memory of the interpreter. This feature is particularly powerful when two programs wish to share information about quantified expressions.

The canonical representation of expressions is an extension of the ATOM feature of LISP. In LISP all strings, including identifiers, that are syntactically equivalent, that is have the same characters in the same order, are converted by the function MKATOM to a pointer to the same word in memory. This word, in turn, gives access to a property list for the atom. In QA4 the constructors for each syntactic type, for example, the primitive programs that make tuples and sets, first

look in memory to see if an equivalent expression has been previously constructed. If so, they return a pointer to that expression; if not, they construct a new expression and make its syntactic component their argument, and finally return a pointer to the new expression. This expression look-up process is explained in detail in Appendix II.

B. Identifiers

An identifier is an individual symbol such as X, Y, MAX, etc. The identifiers are the function symbols, predicate symbols, and variables of the language. Certain identifiers such as AND, OR, and UNION are reserved in that they have predetermined built-in meanings. All other identifiers may generally be used for variables, naming, defined functions, etc.

The rest of this section discusses each of the syntactic types--fully defining the basic ones and outlining the more complex ones such as STATEMENT and LAMBDA.

C. Tuples, Sets, Bags, and Numbers

1. Tuples

A tuple is an expression of the form

(TUPLE _el_ ... _en_)

where _el_ ... _en_ are expressions. Its meaning is the tuple of objects denoted by the expressions. That is, a tuple evaluates to the tuple of the values of its components. For example, (TUPLE 1 1 + 1 6/2)

evaluates to (TUPLE 1 2 3). Two evaluated tuples (TUPLE _e1_ ... _en_) and (TUPLE _f1_ ... _fm_) are equal provided they have the same length (n=m) and each _ei_ is equal to the corresponding _fi_. Because QA4 uses a canonical internal representation, the two tuples will be equal if, and only if, they are the same expression.

In QA4 all functions have exactly one argument. Thus, a tuple is used as the single argument to a function demanding more than one input. The application of a function f to arguments (1 2) (in mathematics f(1 2)) is represented in QA4 as the expression

(APPL f (TUPLE 1 2))

where f takes the tuple (TUPLE 1 2) as its single argument.

In the user language, one may leave out the syntactic type word APPL and merely list the arguments. The QA4 preprocessing system will perform the appropriate translation. Thus, one would write

(F 1 2)

in a QA4 program, but the internal form would be

(APPL F (TUPLE 1 2)) .

2. Sets

A set expression is an expression of the form

(SET _e1_ ... _en_)

where _e1_ ... _en_ are any expressions. The meaning of the set expression is the set of objects denoted by the expressions within the set. Since both the order of the elements and the number of occurrences of an

element in a set are immaterial, the sets

(SET A B C) (SET C A B) (SET C A A B C)

are all treated as identical expressions. When the sets are constructed, either during input or by a program, multiple occurrences of syntactically identical elements are reduced to a single occurrence of the element. Thus, the last set in the above example could never occur internally as a QA4 expression. During evaluation, when the expressions are replaced by the objects they denote, a continued reduction may occur.

3. Bags

A bag expression is an expression of the form

(BAG _el_ ... _en_)

where _el_ ... _en_ are arbitrary expressions. A bag is like a set in that the elements are considered to be unordered. It differs from a set, and resembles a tuple in that elements may have multiple occurrences, for example, the bags (BAG 2 3 2) and (BAG 2 2 3) are equal but neither is equal to the bag (BAG 2 3).

QA4 uses bags or sets as arguments to many primitive functions. Since PLUS is commutative and associative, for example, 2+2+3 can be represented internally as (APPL PLUS (BAG 2 3 2)). The QA4 preprocessor, however, knows the argument type of all primitive functions and will automatically perform necessary reformatting. Thus 2+2+3 would be written in a program as

(PLUS 2 2 3) .

Functions that are also idempotent may take sets as arguments. For example, TRUE & FALSE & TRUE could be represented as (APPL AND (SET TRUE FALSE)) but could be written in a program as (AND TRUE FALSE). Permitting the interpreter to control the order of the elements of bags and sets significantly reduces searching during such jobs as pattern matching and expression simplification.

4. Numbers

A number is simply a LISP number, for example, 3, 0, -17.2 are numbers.

D. Contexts, Processes, and Semaphores

There is no special notation for the syntactic types CONTEXT, PROCESS, and SEMAPHORE. Internal representations for these objects are built and manipulated by primitive functions and statements. Semaphore statements are discussed in Section III-G, Process statements in Section XI, and Context statements in Section VII. The internal form of contexts is described in Appendix III.

E. Applications

An application is an expression of the form (APPL F A) where F is an expression denoting a function and A is any expression. All QA4 functions take one argument; however, the argument can be a tuple, (TUPLE al ... an), so there is no loss of generality. The meaning of an application (APPL F A) is its natural one--namely, the result of applying the function denoted by F to the argument denoted by A. As

we have noted previously, the preprocessor will supply APPL, thus the form in a program is simply (F A). It is not necessary within QA4 to specify the syntactic type of arguments in the function definition. In this way a single program can operate on tuples, sets, or bags. A sort program, for example, could sort either bags or sets into tuples.

F. Bound Variable Expressions

A bound variable expression is an expression of the form (keyword bv e), where e is any expression, bv is a bound variable or pattern, and keyword is one of LAMBDA, FA (for all), or EX (exists). All lambda expressions and quantified expressions are bound variable expressions. The bound variable, bv, is a pattern and has syntax and semantics that extend the usual definition of expressions. It is akin to a variable declaration. Basically, the purpose of a bound variable is to assign values to one or more variables during the evaluation or analysis of the expression e, the body of the bound variable expression.

1. Bound Variables

Unless more advanced pattern-matching features are invoked, a bound variable is usually a normal QA4 expression. Thus the internal and user formats are the same. When the bound variable of a lambda expression is bound or matched to the actual argument for the evaluation of an application, the pattern matcher decomposes the actual argument. As we will see when statements such as SETQ and EXISTS are discussed later, almost all expression decomposition in QA4 programs is accomplished

by pattern matching. The following brief description of patterns should suffice for most examples. The full pattern language is presented in the PATTERNS Section.

a. Pattern Variables

During template pattern matching, variables may take on roles not normally found in programming languages. For example, suppose X has the value 1 prior to the match of (TUPLE X Y) against (TUPLE 2 3). Do we want the match to fail because X already has the value 1? Or should the match succeed with X set to 2? Still another problem with normal variable usage comes from the dominant role of constants in patterns. Variables tend to play a secondary role, and to identify quoted expressions with special syntax, say quote marks, and not to identify the variables obscures the template quality of the patterns.

QA4 uses a quasi-quote mode (Quine, 1965) in all expressions. This means that all symbols are treated as being quoted except for those that are specifically identified as variables. There are many variable identification symbols. Some are meaningful within the context of a single pattern, while others have meaning only in the context of programs. Most examples use only the symbols $ and ←.

- Symbol $

  In user format, a $ immediately preceding an identifier flags it as a variable. The $ must, in fact, be the

first character of the LISP atom that constitutes the identifier. It means that the variable must have a value during evaluation, i.e., be bound to an object. If the variable does not have a value, the program fails. For example, the expression (PLUS 3 $X) with X bound to 2 evaluates to 5, but if X does not yet have a value the program fails. If (TUPLE 3 $X) is matched against (TUPLE 3 2) it succeeds if X had value 2 previous to the match, and it fails if X has any other value or does not have a value.

- Symbol ←

← permits the variable it flags to take a new value within this particular execution of this specific match, but requires that the value be consistent within the match. For example, if we match (BAG ←X ←X) against (BAG 2 2) then if X already has a value it will be ignored and X will take the value 2. But if we match (BAG ←X ←X) against (BAG 2 3) then the match will fail and the value of ←X before the attempted match will be retained. If an expression is being evaluated, then a ← variable must have a value just as a $ variable. For example, to evaluate (PLUS 2 ←X), ←X must have a

value or the expression cannot be evaluated. If the ← variable does not have a value, the program fails.

- Internal Format

The internal format of bound variables is a LISP-style property list that begins (BV _v_ PREFIX _p_) where _v_ is the LISP atom for the variable and _p_ is the prefix code. For example, $X in user format becomes (BV X PREFIX $) in internal format. This compacted form of the user format greatly enhances the template quality of QA4 programs while the internal format eases the tasks of the interpreter in its more complicated operations.

b. Simple Patterns

A pattern with $ variables is a parameterized template for the expression to be decomposed. For example, suppose we wish to match (LEFTOF ←X $Y) to the expression (LEFTOF BLOCK1 TOWER3) and $Y is bound to TOWER3. First the pattern is instantiated. That is, all $ variables are replaced by their values and the template is constructed. In our example (LEFTOF ←X TOWER3) would be the template. The internal format of the expressions would now be (APPL LEFTOF (TUPLE ←X TOWER 3)) and (APPL LEFTOF (TUPLE BLOCK1 TOWER3)). The pattern matcher must check that the same constants appear in the same positions, construct

a binding list for newly encountered variables, and assure the consistency of the role of variables on later encounters.

If the pattern contains a set or bag of bound variables, then the expression to be decomposed must have an expression of the same syntactic type in the same position. Here the situation is more complex since there is more than one possible assignment. An initial assignment is chosen by the pattern matcher, and a backtracking point is established by the interpreter. Future failures may return to the backtracking point, at which time alternate assignments are made. For example, we may wish to bind (NEXTTO (BAG ←X ←Y $Z)) to (NEXTTO (BAG BLOCK1 BLOCK2 TOWER3)) with $Z bound to BLOCK2. The pattern matcher may choose an initial binding of ←X to BLOCK1 and ←Y to TOWER3. Later, if the program fails back to this point, the alternate binding of ←X to TOWER3 and ←Y to BLOCK1 would be established. A second failure to this point would cause a failure from this backtracking point to its immediate predecessor, for there are only two possible ways to carry out this match.

2. Lambda and Quantified Expressions

A LAMBDA expression is a bound expression with both user and internal form (LAMBDA bv e). A LAMBDA expression denotes a function whose value at an argument A is the value of e with the variables of e set to the result of binding bv to A. The main use of LAMBDA expressions is in defining new functions. An example is

(LAMBDA (TUPLE ←X ←Y) (TUPLE $Y $X)) .

This is a function that reverses a tuple of length 2. (TUPLE ←X ←Y) is the _bv_, and (TUPLE $Y $X) is the expression body _e_.

If _bv_ contains a set or bag that, in turn, contains a variable there may be more than one possible binding. For example,

(LAMBDA (BAG ←X ←Y) (TUPLE $X $Y))

applied to (BAG 1 2) could evaluate to either (TUPLE 1 2) or (TUPLE 2 1). For such applications the interpreter _does not_ normally establish a backtracking point and performs the alternate bindings and evaluations on failure. It uses the first binding chosen by the pattern matcher. Since it does not establish a backtracking point, a failure bypasses this potential choice point. If the user wishes the interpreter to invoke alternate bindings on failure, an alternative form of the lambda expression must be used:

(LAMBDA _bv_ _e_ BACKTRACK) .

A _quantified expression_ is an expression of the form (_quantifier_ _bv_ _e_), where _e_ is a truth-valued expression and _quantifier_ is either the universal quantifier FA (for all) or the existential quantifier EX (exists). The mathematical (∀X)(∀Y)(∀Z)P(X Y Z) is expressed in QA4 as

(FA (TUPLE X Y Z) (P X Y Z)) .

We have, at times, considered adding other quantifiers to the QA4 language. This will probably be done, but only as the need arises

in the construction of problem-solving programs. Some candidates are:

- The X such that F(X), sometimes known as the denotation or choice operator.
- There is exactly one X such that F(X).
- The set of objects X such that F(X), an extended denotation operator.

Within QA4 there is no notion of an application of a quantified expression to an argument (there is, however, the application of lambda expressions to arguments).

3. * Extended Applications

a. The WITH Clause

We have designed, but not yet implemented, an extension of applications that permits the programmer to specify strategies to guide the pattern matcher during the bound variable decomposition. For example, suppose we define a function FUN to be

(LAMBDA (BAG ←X ←Y) (TUPLE $X $Y)) .

In normal applications we have no way to force the pattern matcher to choose one particular order over another. To force an order we must use a WITH clause as an option in the application:

($FUN (BAG 1 2) WITH $MYORDER) .

MYORDER is a function we have defined that will communicate with the interpreter and thus force an order on the possible choices. It must have a 2-tuple as its <u>bv</u>. This 2-tuple has as elements

- A pattern that will be bound to the bv of FUN; and
- A pattern that will be bound to the actual arguments to which FUN is applied.

We may have defined MYORDER as

```
(LAMBDA (TUPLE (BAG ←U ←V)
               (BAG ←A ←B))
        (PROG  ( )
           (IF (LT $A $B) THEN (SETQ (TUPLE ←B ←A)
                                     (TUPLE $A $B)))
           (BIND $U $A $V $B)
           (BIND $U $B $V $A)))
```

.

Now when the interpreter begins to apply FUN to (BAG 1 2) and encounters the WITH clause it calls MYORDER rather than the pattern matcher. The argument it constructs for MYORDER is the 2-tuple (TUPLE (BAG ←X ←Y) (BAG 1 2)). The first element of this 2-tuple is the bv of FUN and the second element of the 2-tuple is the actual argument for this application of FUN. That is, the bv of MYORDER is decomposed by matching

(BAG ←U ←V) to (BAG ←X ←Y)

which results in binding

←U to ←X and

←V to ←Y

and by matching

(BAG ←A ←B) to (BAG 1 2)

which results in binding

←A to 1 and

←B to 2 .

At this point MYORDER is run as a process. When it executes a BIND statement, the specified bindings are established for FUN and FUN is started. The arguments of the BIND statement are an alternating series of variables that are to be bound and the corresponding values. The interpreter establishes the bindings and suspends the WITH program as though a RESUME statement had been executed. In our example X is bound to 2 and Y is bound to 1. If a failure occurs that would normally backtrack to try an alternate binding for FUN, MYORDER is resumed instead. Our example would then bind X to 1 and Y to 2. When MYORDER initiates a return, either through a RETURN statement or by completing a PROG, no more BIND statements can be executed and a failure is created.

Thus a user strategy has replaced the normal function of the pattern matcher. This strategy may, however, rely heavily on the pattern matcher for its internal operation, but order the bindings in a way that would work better than the ordering the system would produce on its own.

b. The ALL Clause

The ALL clause acts as an iterative operator for applications. It serves the same purpose as the MAP functions in LISP. For example,

((BAG $F1 $F2) A ALL FNS)

means to form the bag of applications

(BAG ($F1 A) ($F2 A)) .

The form

($F1 (BAG A B) ALL ARGS)

on the other hand, means to form the bag

(BAG($F1 A) ($F1 B))

rather than to apply F1 to (BAG A B) as an argument.

((BAG $F1 $F2) (BAG A B) ALL FNS ALL ARGS)

means to form the bag of all four possible applications.

ALL may be applied to sets and tuples as well as bags. This feature is not implemented, but appears to have potential value in some problem-solving programs, especially when used with REPEAT statements. It can easily be added when the need arises.

G. Statements

In QA4, statements are syntactic forms that violate the standard rules of expression evaluation. The normal mode of evaluation of applications in QA4 is to first evaluate the operator (function part), then the operand (argument part), do the binding, and then evaluate the body of the function. Many times, however, concise syntactic forms require lists of directions or options and must bypass the normal mode of evaluation. The most common example of this is the IF or COND form.

ALL syntactic forms that have nonstandard evaluation rules have syntactic-type STATEMENT, and are the QA4 statements.

As we have seen, QA4 operates in inverse quasi-mode where all symbols are quoted except those flagged as being variables. This representation gives statements an especially natural appearance. Another addition to this natural syntax is the instantiation rather than evaluation of some parts of statements. That is, the expression (PLUS $X $Y), with $X bound to 3 and $Y bound to 2, stands for the expression (PLUS 3 2) if it occurs as a part of a statement that is instantiated rather than evaluated.

The AMONG statement is a simple example of the way statements are interpreted. AMONG takes a series of expressions and has as its value the value of one of the expressions. For example,

(AMONG 1 2 3)

will have the value 1, 2, or 3. The internal form of the statement is

(STATEMENT AMONG 1 2 3)

but the preprocessor always supplies the type word STATEMENT.

When the interpreter executes the statement it instantiates only one of the expressions, establishes a backtracking point, and proceeds with the program. If the program backtracks to this statement, another expression is instantiated, and so on. If the expressions are finally exhausted the statement fails, and forces backtracking to the previous backtracking point.

Notice that as each expression is chosen as a candidate for the value of the statement it is not evaluated in the normal way but merely instantiated. That is, an expression is formed that is just like the expression in the AMONG statement except that all the $ variables have been replaced by their values. For example, if X, Y, and Z have values 1, 2, and 3 respectively,

```
(AMONG (PLUS $X $Y) (PLUS $Y $Z))
```

would have either the expression (PLUS 1 2) or the expression (PLUS 2 3) as its value (not 3 or 5).

III PRIMITIVE DATA OPERATIONS

The primitive QA4 operations can be broadly separated into six categories: logical (AND), structural (UNION), arithmetic (PLUS), constructors (CONS), decomposers (NTH), and syntactic information (STYPE).

A. Logical Operators

- (AND pl ... pn)

  The operator AND takes a set of truth values and returns TRUE if FALSE is not a member of the set and FALSE if otherwise. Thus, (AND pl ... pn) is true provided none of the expressions denotes FALSE.

- (OR pl ... pn)

  OR is analogous to AND except that it returns TRUE if any pi denotes any expression other than FALSE. That is, (OR pl ... pn) is FALSE only if pl ... pn all denote FALSE.

- (EQUAL el ... en)

  EQUAL is TRUE if all of the members of the set are logically equal--that is, denote the same object.

- (NOTEQUAL el ... en)

  NOTEQUAL is TRUE if there are at least two different objects denoted by the expressions el ... en. Equal can be considered as a conjunct of pairwise EQUAL assertions. Thus, NOTEQUAL

can be considered the corresponding disjunct. That is

(EQUAL _e1_ _e2_ _e3_) is equivalent to

(AND (EQUAL _e1_ _e2_) (EQUAL _e1_ _e3_) (EQUAL _e2_ _e3_))

while

(NOTEQUAL _e1_ _e2_ _e3_) is equivalent to

(OR (NOT (EQUAL _e1_ _e2_)) (NOT (EQUAL _e1_ _e3_)) (NOT (EQUAL _e2_ _e3_))) .

Since AND, OR, EQUAL, and NOTEQUAL are commutative, associative, and idempotent (e.g., (AND A A) is equivalent to A), they have been given sets as arguments rather than bags or tuples. Thus their internal forms are:

(APPL AND (SET _p1_ ... _pn_))

(APPL OR (SET _p1_ ... _pn_))

(APPL EQUAL (SET _e1_ ... _en_))

(APPL NOTEQUAL (SET _e1_ ... _en_)) .

- (NOT _p_)

  NOT negates the truth value of its argument. It is FALSE if its argument is any expression other than FALSE, and TRUE only if its argument is FALSE.

- (IMPLIES _p1_ ... _pn_)

  This form of IMPLIES corresponds to the mathematical form

  $p1 \rightarrow p2 \rightarrow p3 \ldots \rightarrow pn$

  which is also equivalent to the QA4 form

(AND (IMPLIES _p1_ _p2_) (IMPLIES _p2_ _p3_) ... (IMPLIES _pn-1_ _pn_)) .

For it to be TRUE all the _pi_'s that denote FALSE must come before any that denote TRUE. Since QA4 uses the convention that any expression other than FALSE denotes TRUE, IMPLIES is TRUE unless an expression denoting FALSE follows an expression not denoting FALSE.

- (IFF _e1_ ... _en_)

  IFF is TRUE if every member of the set is not FALSE. IFF like EQUAL, takes a set as its argument.

B. Structural Operations

- (IN _x_ _s_)

  IN is TRUE if _x_ is an element of _s_. _s_ may be either a set, bag, or tuple.

- APPEND

  APPEND performs three different operations depending on the syntactic type of its argument. If the argument is a tuple of subtuples it joins together the subtuples, if it is a bag of subbags APPEND adjoins the subbags and may reorder the result; if it is a set of subsets APPEND takes the set-theoretic union of the subsets. For example,

  (APPEND (TUPLE (TUPLE 4 3) (TUPLE 4 3 2 1)))

  = (TUPLE 4 3 4 3 2 1)

(APPEND (BAG (BAG 4 3) (BAG 4 3 2 1)))

= (BAG 1 2 3 3 4 4)

(APPEND (SET (SET 4 3) (SET 4 3 2 1)))

= (SET 1 2 3 4) .

APPEND depends on the syntactic type of its argument which may vary, the preprocessor will not automatically group together arguments--the user form and the internal form are the same.

- (INTERSECTION _s1_ ... _sn_)

  INTERSECTION is the set-theoretic intersection of the sets _s1_ ... _sn_. INTERSECTION takes a set as its argument, so the internal form is

  (INTERSECTION (SET _s1_ ... _sn_)) .

- (DIFFERENCE _s1_ ... _sn_)

  DIFFERENCE is the set-theoretic difference of the two sets _s1_ and (APPEND (SET _s2_ ... _sn_)). Since the order of the _s_'s is important, DIFFERENCE takes a tuple as its argument. The internal form is

  (DIFFERENCE (TUPLE _s1_ ... _sn_) .

- (CONS _x_ _e_)

  CONS adjoins the element _x_ to the tuple, bag, or set _e_. For example,

(CONS 1 (TUPLE 1 2 3)) = (TUPLE 1 1 2 3)

(CONS 1 (BAG 1 2 3)) = (BAG 1 1 2 3)

(CONS 1 (SET 1 2 3)) = (SET 1 2 3) .

CONS takes a tuple as its argument, thus the internal form is

(CONS (TUPLE x e)) .

C. Arithmetic Operations

- (PLUS nl ... nm)

  PLUS forms the sum of the elements nl ... nm. It takes a bag as its argument, so the internal form is

  (PLUS (BAG nl ... nm)) .

- (TIMES nl ... nm)

  TIMES forms the product of the elements nl ... nm. It also takes a bag as its argument, so the internal form is

  (TIMES (BAG nl ... nm)) .

- (SUBTRACT nl ... nm)

  SUBTRACT subtracts the sum of n2 through nm from nl. It is equivalent to

  (SUBTRACT nl (PLUS n2 ... nm)). It takes a tuple as its argument, so the internal form is

  (SUBTRACT (TUPLE nl ... nm)) .

- (DIVIDES nl ... nm)

  DIVIDES forms the quotient of nl and the product of n2 ... nm. Thus DIVIDES is equivalent to

(DIVIDES _n1_ (TIMES _n2_ ... _nm_)) .

It takes a tuple as its argument, so the internal form is

(DIVIDES (TUPLE _n1_ ... _nm_)) .

- (MINUS _n_)

  MINUS forms the arithmetic negation of its single argument.

- All the arithmetic relation operators take a tuple as their argument. The tuple means that the relation must hold for each juxtaposed pair of the tuple. That is, if _r_ is a primitive arithmetic relation,

  (_r_ _n1_ _n2_ ... _nm_) means

  (AND (_r_ _n1_ _n2_) (_r_ _n2_ _n3_) ... (_r_ _nm-1_ _nm_)) .

  Since the preprocessor will supply the TUPLE, the internal form of

  (_r_ _n1_ ... _nm_) is

  (_r_ (TUPLE _n1_ ... _nm_)) .

  The relation symbols and their meanings are:

  GT greater than

  LT less than

  GTQ greater than or equal to

  LTQ less than or equal to .

D. Constructors

The most commonly used constructors in the system are TUPLE, SET, and BAG. Every time a TUPLE is evaluated, for example, a new TUPLE is

constructed from the values of its elements. In a similar sense APPEND, INTERSECTION, DIFFERENCE, and CONS are also constructors. While they are primitive operators, they are not, however, primitive constructors for internally they rely on the primitives TUPLE, SET, and BAG.

1. (QUOTE *e*)

QUOTE is the simplest constructor, for *e* is its value. For example, (QUOTE (SET ←X ←Y)) evaluates to (SET ←X ←Y).

2. (' *e*)

' is a special form of QUOTE, called quasi-quote, added to QA4 to cause instantiation in places where evaluation would normally take place. For example, suppose $X is bound to BLOCK1, and we wish to call function FUN with the instantiated form of

```
(NEXTTO (SET ROBOT $X))    .
```

If we write

```
($FUN (QUOTE (NEXTTO (SET ROBOT $X))))
```

in our program then $X will not be replaced by BLOCK1 in the argument to FUN. If we write

```
($FUN (NEXTTO (SET ROBOT $X)))
```

then the interpreter will attempt to evaluate NEXTTO applied to (SET ROBOT BLOCK1) and use the result as the argument to FUN. Since NEXTTO may not have a function definition, an error may result.

The ' operator provides the mechanism for constructing the argument properly. When ' is applied to its argument, the value is the instantiated form of the argument. That is, all $, ?, and fragment variables are replaced by their values. For example,

```
(' (NEXTTO (SET ROBOT $X)))
```

gives

```
(NEXTTO (SET ROBOT BLOCK1))   .
```

Thus the proper notation for the function call would be

```
($FUN (' (NEXTTO (SET ROBOT $X))))   .
```

The section on PATTERNS gives a full explanation of this process.

The ' will construct QA4 expressions of all syntactic types, but there is still the problem of constructing an expression with a variable in it. Suppose, for example, we wish to apply a function, say FUN to the expression 1 + $X where the $X is meant literally and not to be replaced by its value. If we precede the $, ←, or ? of a variable immediately with a :, the instantiated form will contain the variable without the :. Actually, any number of :s can be used and one less will appear in the instantiated form. For example,

```
$X gives $X
:::?Y gives ::?Y   .
```

Thus the form for our application will be

```
($FUN (' (PLUS 1 :$X)))   .
```

The internal form of a variable with a : is

(BV v PREFIX pfx : n)

where v is the LISP atom for the variable, pfx is the $, ?, or ← prefix symbol, and n is the number of :s.

E. Syntactic Information

The primitive function STYPE returns the syntactic type of an expression. For example,

(STYPE (' (PLUS 1 2)))

is APPL, and

(STYPE (' (AMONG 1 2 3)))

is STATEMENT.

Currently, this is the only function that returns syntactic information. If functions such as LENGTH are added they will be in this class.

F. Decomposition: (NTH t n)

NTH extracts the nth element from a tuple. For example,

(NTH (TUPLE 1 2 3) 2) = 2 .

NTH is the only primitive function for expression decomposition. Expressions are normally taken apart and their components named through the use of the pattern matching language. Most of the pattern matching within a program arises from one of the following language usages:

- LAMBDA expressions that take a single argument which is decomposed according to a pattern.
- Assignments made by assigning an expression to a pattern.
- WHEN and GOAL statements that use patterns to guide the program flow of control.
- EXISTS and INSTANCES statements that use patterns to query the data base for expressions that match their patterns.

G. * Semaphores

Semaphores are a syntactic type that has not yet been implemented. They would be used to synchronize parallel programs. The primitive operations on them would be:

- (SEMAPHORE v)

  Creates a new semaphore and assigns it to the variable v.
- (SEMAPHORE-SET v)

  Turn on the semaphore assigned to v.
- (TEST-IF-SET v)

  Has value TRUE only if v is on; otherwise it has value FALSE.
- (WAIT-UNTIL-OFF v)

  Suspend this process until the semaphore assigned to v becomes off.

The assignments of semaphores to variables operate in an unusual way. Suppose $X is bound to a semaphore. If the operation of calling a function binds the value of $X to, say, $Y, the binding is like a

"call by name." If $Y is reset, the semaphore itself is changed, not the value of $Y. In this way $X also changes appropriately. This happens because semaphore statements do not bind Y or X to new values, but rather change the semaphore itself. Thus if X and Y are both bound to the same object, changing the object changes them both.

IV PATTERNS

A. Motivation

The pattern language plays an important part in nearly every aspect of QA4, making it possible to describe actions in a single, often graphical way. For example, if we match (TUPLE ←X ←Y ←←Z) against (TUPLE 1 2 3 4 5) the assignments are 1 to X, 2 to Y and (TUPLE 3 4 5) to the fragment Z. In LISP three assignments using operations such as CADR and CDDR would be equivalent to this single QA4 assignment. As another example,

(LAMBDA (SET (TUPLE 1 ←X) ←←Z) body)

is a lambda expression that expects as an argument

- A set
- With a 2-tuple as one of its elements
- With 1 as the first element of the 2-tuple.

The variable X will be bound to the second element of the 2-tuple, and the variable Z will be bound to the remaining elements of the set. If there is more than one 2-tuple with 1 as the first element in the set, there will be more than one acceptable assignment. If there are no such 2-tuples, the application will fail. These two examples show the simplicity of a pictorial description of data structure decomposition.

In general the pattern language and matcher is used for locating subexpressions in a larger expression and perhaps naming the located subexpressions and expression transformations. Operations of this sort

are difficult to describe in a function-oriented language. In LISP, the absence of a pattern matcher leads to indigestible programs that are written with many CADDRs, CONSs, LISTs, and APPENDs.

Use is often made of partially specified patterns. In the EXISTS statement, for example,

(EXISTS PROVE (ON GREENBLOCK ←X))

the system will try to find an item in the data base that matches the pattern (ON GREENBLOCK ←X). Such an item could be (ON GREENBLOCK REDBLOCK).

Still another application is the use of patterns to specify the demons of WHEN statements. The WHEN statement is activated whenever an expression in the data stream that the WHEN is watching matches the pattern given in the WHEN statement. For example,

(WHEN X RECEIVES (TUPLE ←←Z 7) THEN p)

will trigger action p each time a tuple with last element 7 is assigned to the variable X.

B. Constants

An atom without a prefix is treated as a constant and will only match another instance of itself. For example,

(NEXTTO (BAG ROBOT BLOCK1))

matches itself, and would therefore match

(NEXTTO (BAG BLOCK1 ROBOT))

but will not match either of the following:

(NEXTTO (TUPLE ROBOT BLOCK1))

(NEXTTO (BAG ROBOT BLOCK2))

Since QA4 uses canonical forms for expressions, only one instance of an expression can actually occur in the QA4 memory, and constants match only if they are the same constant.

C. Variables

If an atom is to be treated as a variable, it must have a prefix. There are six variable prefixes: ←, ?, $, ←←, ??, and $$. The first three prefixes restrict the variable to match only individual terms. The double-character prefixes allow the variables to match "fragments" or segments, but otherwise play analogous roles to their single-character counterparts. In user format, the prefix must be the initial characters of the LISP atom that constitutes the variable. See Section II-F-1-a, Pattern Variables, for an introductory explanation of the use of prefixes.

1. ← (Accept a Value)

The ← prefix will permit a variable to match any individual term, regardless of whatever QA4 value the variable may have. For example,

(TUPLE 1 ←X 2)

matches

(TUPLE 1 (BAG 1 2) 2)

by matching ←X to (BAG 1 2). And there are two ways

(BAG ←X ←Y)

matches

(BAG 1 2)

←X may match 1 and ←Y may match 2, or vice versa.

Note, however, that the matching of different instances of the same variable within a single pattern must be consistent. For example, there is only one way

(BAG ←X ←Y ←X)

matches

(BAG 1 1 2)

with ←X matched to 1 and ←Y matched to 2.

2. <u>? (Accept a Value If You Can)</u>

A ? variable will behave differently from an ← variable if the variable had a QA4 value before the match was attempted. In that case, the variable will only match its value; otherwise the ? variable will behave like an ← variable and match any term, proving the matching is consistent with other previous matches the variables may have. For example, if X has value 2,

(TUPLE 1 ?X 3)

matches

(TUPLE 1 2 3)

but does not match

(TUPLE 1 1 3) .

Note that, even if X does not have a value,

(TUPLE ←X ?X)

will not match

(TUPLE 1 2) .

3. $ (Has a Value)

If the variable has no value, it will not match anything and will cause a failure. If it does have a value, it will only match that value. We use $ variables mainly when we expect the variable to have a value and do not want it to be bound to anything else as a result of the match. For example, if X has value 1,

(TUPLE $X 2)

matches

(TUPLE 1 2)

but would not match

(TUPLE 2 2) .

4. Summary of Prefix Types

We may summarize the differences between the prefixes in the following table representing the result of matching the variable X against the constant A. The vertical axis represents the prefix of X, and the horizontal axis represents the QA4 value of X.

The item in the table represents the value of X after the match; "unbound" means it has no value, and NIL means the match failed. If a match fails, the variable always has the value it had previous to the attempted match.

X matched against A

	A	unbound	B
←X	A	A	B
?X	A	A	NIL
$X	A	NIL	NIL

5. Notes on Matching Sets and Bags

Sets and bags may have their elements rearranged in searching for a match. Thus,

(SET ←X A ←Y)

can match

(SET A B C) .

Furthermore, more than one distinct element of a set pattern can match the same element of the argument. For instance,

(SET ←X ←Y B)

will match

(SET B)

with X matched to B and Y also matched to B. However, the same is not true of bags. For example,

(BAG ←X ←Y) .

will not match

(BAG A) .

This convention reflects the difference between the set and bag concepts: equal elements are identified in sets but not in bags.

Nondeterminism occurs in matching sets and bags. For instance, matching

(SET ←X ←Y)

against

(SET A B)

can produce either

X matched to A

Y matched to B

or

X matched to B

Y matched to A .

In practice, one of these will be produced on the first match, and information to proceed in future matches will be kept by the backtracking mechanism of the interpreter. A failure to this backtracking point will cause the interpreter to call the pattern matcher for alternative bindings. When all alternatives are exhausted, a failure to this backtracking point will cause a failure to the preceding backtracking point.

6. Fragments

Variables prefixed with ←←, ??, or $$ match not single items but fragments of the argument. For example,

(TUPLE ←←X 3)

can match

(TUPLE 1 2 3)

with X matched to (TUPLE 1 2). X is thought of as having matched the sublist (1 2), but that list is put into a tuple for the sake of consistency of internal representation. Fragment variables may match the empty fragment. For example,

(TUPLE ←←X A)

will match

(TUPLE A)

with X matched to the empty tuple (TUPLE). Fragment patterns of sets and bags will be bound to sets and bags respectively, and order need not be preserved. For example,

(SET ←←X B)

matches

(SET A B C)

with X matched to (SET A C). Sets and bags may have only one fragment variable; however, tuples may have more than one, and this leads to nondeterminism. For example,

(TUPLE ←←X ←←Y)

can match

(TUPLE A B)

in three different ways:

(1) X matches (TUPLE)

Y matches (TUPLE A B)

(2) X matches (TUPLE A)

Y matches (TUPLE B)

(3) X matches (TUPLE A B)

Y matches (TUPLE) .

A single variable may be prefixed in several ways in a single pattern. For example,

(TUPLE ←←X ←X)

can match

(TUPLE A (TUPLE A)) .

Variables may also match parts of expressions other than tuples, sets, and bags. For example,

(←F (BAG ROBOT ←X))

can match

(NEXTTO (BAG ROBOT BLOCK1))

with F matched to NEXTTO and X matched to BLOCK1.

The fragment prefixes ?? and $$ behave analogously to the individual prefixes ? and $; e.g., a variable prefixed $$ must have a value already, and must match a fragment equal to its value.

Patterns may be nested to arbitrary depth. Thus,

(PLUS (BAG (EXPT (COS ←X) 2) (EXPT (SIN ←X) 2)))

matches

(PLUS (BAG (EXPT (SIN 30) 2)(EXPT (COS 30) 2))) .

The pattern is one side of a well-known trigonometric identity. As another example,

(SET ←←X (SET (NOT ←P) ←←Q) (SET ←P ←←R))

matches

(SET (SET A B C) (SET (NOT A) B D) (SET E)) .

The above pattern might be the argument for a function representing the propositional calculus resolution rule.

D. <u>Instantiation</u>

An introductory explanation of instantiation is given in Sections II-G (Statements) and III-D (Constructors).

The rules are straightforward. For the following examples, suppose X has value 1; Y has value (TUPLE 3 4); and Z has no value.

- Constants remain as constants.
- $ variables are replaced by their values.

  (TUPLE $X 2) becomes (TUPLE 1 2)

  (TUPLE $Y 2) becomes (TUPLE (TUPLE 3 4) 2)

  (TUPLE $Z 2) causes a failure.

- ? variables are replaced by their value if they have one. If they lack a value they are converted to ← variables.

  (TUPLE ?X 2) becomes (TUPLE 1 2)

  (TUPLE ?Z 2) becomes (TUPLE ←Z 2) .

- ← variables remain as they are.

  (TUPLE ←X 2) becomes (TUPLE ←X 2) .

- Fragment variables follow the same rules except that when they are replaced by their value, they are expanded into a sublist rather than maintaining the status of a single item.

  (TUPLE $X 2 $$Y) becomes (TUPLE 1 2 3 4)

  (TUPLE $Y $$Y ??Z) becomes (TUPLE (TUPLE 3 4) 3 4 ←←Z) .

E. Extended Constructions

These features use special operators to signal either the instantiation, matching, or lambda binding programs that extraordinary operations are to take place. These patterns follow the same syntax as regular QA4 expressions, but they use operators that have a meaning only in some phase of the pattern matching process. These are the exceptions to the template rule--that expressions and patterns are identical.

1. Expression =

When = is applied to an expression, the instantiation program will evaluate the expression and use the value as the pattern. For example,

```
(= (TUPLE ←X (PLUS 1 2)))
```

instantiates to

```
(TUPLE ←X 3)    .
```

Note that the internal form for the uninstantiated pattern would be

```
(APPL = (TUPLE ←X (PLUS (BAG 1 2))))    .
```

The = operator may only be used on the top-level of a pattern. While the expression is being evaluated, the interpreter guards against backtracking out of the evaluation. If a backtracking attempt is made, the program proceeds to the next statement. For example,

```
(SETQ (TUPLE ←X ←X) (TUPLE 1 2))
```

has no effect whatever. This is similar to the protection offered in the ATTEMPT statement.

2. Expression ..

A pattern of the form .. pat, where pat is a pattern expression matches an argument if pat matches some subexpressions of that argument, perhaps the entire argument itself. For example,

```
( .. (PLUS (BAG ←←X 0)))
```

matches

```
(TIMES (BAG A (PLUS (BAG 0 B C))))
```

by matching X to (BAG B C).

The .. operator and its argument need not be an application if they are embedded in another expression. The operator is assumed to take the expression immediately to its right. For example,

(.. (←P ←←X) .. (NOT ←P ←←Y))

matches

((AND (BETWEEN C A B) (ONTOP C T))

(IF (NEAR C D) (NOT (BETWEEN C U V)))) .

3. *Logical Combinations of Patterns

- PAND

Several patterns can be combined with a Pattern AND. It simply means that the argument must match all the given subpatterns. A common use of this feature is to name an expression and decompose it in a single operation. For example,

(PAND ←X (TUPLE 1 ←Y))

matches

(TUPLE 1 2)

by matching X to (TUPLE 1 2) and Y to 2. The matching of variables within a PAND must be consistent, just as they must be for a normal pattern.

- POR

The Pattern OR succeeds if one of the patterns matches the argument. For example,

(POR (TUPLE 1 ←X 2) (TUPLE 1 ←X 4))

matches

(TUPLE 1 3 4)

with X matched to 3. Both PAND and POR applications are done by the pattern matcher directly, rather than during instantiation. Thus they

may occur at any level in the pattern. For example,

```
(PAND ←X (POR (TUPLE 1 ←Y) (TUPLE 2 ←Y)))
```

matches

```
(TUPLE 2 3)
```

by matching X to (TUPLE 2 3) and Y to 3.

4. *Type Constraints*

A variable's domain may be constrained by providing syntactic type information. For example, ←X/SET can only have sets assigned to it. The / and the type word must be part of the ATOM that constitutes the variable. The internal form is expanded to include this property. The checks must be made in the pattern matcher, when it associates a value with the variable.

5. *Predicate Constraints*

With each pattern can be associated an evaluable QA4 expression that can constrain the match. The predicate may use the variables in the match, but if the match fails or the predicate is not true, the variables will be restored to their previous values. The predicate operator is ↑. For example,

```
(↑ (TUPLE 1 ←X) (LT $X 2))
```

matches

```
(TUPLE 1 0)
```

but would not match

```
(TUPLE 1 2)
```

.

This feature can be implemented by building T:MATCHO to watch for ↑ on the top level, and do the evaluation after the match. The evaluation should be protected against backtrack failures in the same way = is protected. On a predicate failure with a potential alternative match, a rematch should be tried immediately. If no match works, T:MATCHO can return its usual NIL.

F. Internal Representation

The preprocessor translates the user form into an internal form. A table of representative sample expressions together with translations is given.

```
User                          Internal

←X                            (BV X PREFIX ←)

: ←X                          (BV X PREFIX ← : 1)

?X/SET                        (BV X PREFIX ? STYPE SET)

(= ($F $A))                   (APPL = (APPL (BV F PREFIX $)
                                            (BV A PREFIX $)))

(PAND ←X (TUPLE 1 ←Y))        (APPL PAND (SET
                                 (BV X PREFIX ←)
                                 (TUPLE 1 (BV Y PREFIX ←))))

(TUPLE 1 .. (BAG 1 ←Y))       (TUPLE 1
                                 (PAPPL OCCURSIN (BAG 1
                                               (BV Y PREFIX ←))))
```

V PROPERTIES

A. General Statements

The PUT, GET, and ERASE statements manipulate the properties of expressions. The forms are:

(PUT pexp ind prop ctx-rec)

(GET pexp ind ctx-rec)

(ERASE pexp ind ctx-rec) .

In each statement pexp, ind, and prop are QA4 patterns and are instantiated rather than evaluated; ctx-rec are options that will be discussed later. A PUT statement might be

(PUT $E P5 $X) .

This sets the value of indicator P5 to the value of X on the expression that is the value of E.

It is vital, at this point, to understand exactly under which context the changing takes place. We assume that normally the program is generating a global data base, and that assertions and properties are to be made available to all programs, not just the current one. Thus, all property manipulation statements are done with the most global dynamic context and the current backtracking context. See Appendix III for a detailed explanation of these terms. Using these two contexts permits all parallel process and backtrack bookkeeping to be handled automatically while all properties are available to any program regardless of the current function nesting. If a GOAL or CASE statement is

in effect, and it had a WRT option, then the context specified in that option is used instead of the most global context (see the CASES statement).

Programs may create their own contexts as they wish, and thus override the system. This permits the programs to perform hypothetical reasoning outside the framework of function nesting and its corresponding variable binding. If the programs want to assign properties relative to other contexts, say during the solution of a frame-problem or conditional derivation, they may specify it under ctx-rec.

A PUT statement may not change an existing property unless that specific option is requested. It also activates all appropriate WHEN statements unless a ctx-rec option states otherwise. The value of a PUT statement is the instantiated prop.

The GET statement retrieves the property under the same context used by the PUT statement. As with the PUT, all WHEN programs that apply are activated. As we will see in the WHEN discussion, each WHEN program may specify whether it is to be activated on PUTs, GETs, ERASEs, or any combination of them. The ERASE statement also uses the global context and activates WHEN statements.

Certain special properties are available to the user, but they must not be changed or QA4 will stop working correctly. Currently the indicators for these are EXPV (the syntactic form of the expression) and INDEX (the set-bag ordering code).

B. Macro Statements

Other properties used by the QA4 interpreter may be manipulated by the user. The indicator MODELVALUE is normally used in QA4 programs to model an environment. The ASSERT and DENY statements put TRUE and FALSE respectively as the properties of this indicator. They use the same context as the PUT statement. Their form is:

(ASSERT pexp ctx-rec)

(DENY pexp ctx-rec) .

The SETQ statement evaluates an expression, exp, and then matches it to a pattern, pexp. If the value of exp, call it vexp, can be decomposed into the form pexp, all the variables within pexp will have their value set to the corresponding part of vexp. The form of the statement is

(SETQ pexp exp ctx-rec) .

If the match fails, the statement fails and the program backtracks to the most recent backtracking point. If the match is nondeterministic, a backtracking point is not established unless an alternative form of the statement is used:

(SETQ pexp exp BACKTRACK ctx-rec) .

If this form is used and the match is nondeterministic, a backtracking point is established and alternative matches are tried if the program fails back to this statement.

The SETQQ statement operates in a similar way, except that it does not evaluate _exp_, but instead uses the instantiated form. For example,

```
(SETQ ←X 3)
(SETQ ←Y 5)
(SETQ ←U (PLUS $X $Y))
(SETQQ ←V (PLUS $X $Y))
```

sets the value of U to 8 and the value of V to (PLUS 3 5).

C. Equivalence Relations

1. The Forms

The EQUIVASSERT and EQUIVDENY statements are automatic mechanism for storing and manipulating information about arbitrary equivalence relations. The forms of the statements are:

(EQUIVASSERT _r_ _rbar_ _e1_ _e2_ ... _en_ _ctx-rec_)

(EQUIVDENY _r_ _rbar_ _u1_ _u2_ ... _un_ _ctx-rec_)

where _r_ and _rbar_ are indicators used to stand for the relation and its negation (say EQUAL and UNEQUAL). Each _ei_ and _ui_ is an expression and _ctx-rec_ is the standard context recommendation option available on all the property manipulation statements. This discussion will carry through a single, detailed example illustrating the full operation of the statements. To begin, the statement

(EQUIVASSERT EQUAL UNEQUAL A B C)

asserts to the system that A=B=C, while

(EQUIVDENY EQUAL UNEQUAL F G H)

asserts that F, G, and H are not all equal. That is, they do not all have the same value. Logically this is equivalent to

$$(F \neq G \text{ or } F \neq H \text{ or } G \neq H) \quad .$$

These statements operate by manipulating sets of expressions stored under the indicators <u>r</u> and <u>rbar</u> on each of the mentioned expressions. They do not operate by storing the value TRUE on expressions such as

(EQUAL (SET A B C)) .

This results in enormous savings in both time and space. It also permits the system to do a complete consistency check of the assertions, and generate a failure if an inconsistency occurs. Thus the user is encouraged to use these statements rather than the equal operator.

Throughout the rest of this discussion we will use EQUAL and UNEQUAL as <u>r</u> and <u>rbar</u>; but other indicators may be used, and more than one relation may be handled simultaneously.

2. <u>Partition Sets</u>

Before we examine the operation of the statements let us examine the data structures and their interpretation. Suppose we have some finite collection of expressions, say

A, B, C, D, E, F, G, H, I .

If we declare some set of them to be equal with the EQUIVASSERT statement the equality is represented as sets (or equality partitions) stored as properties on each of the expressions. For example, if we

start with a fresh system

```
(EQUIVASSERT EQUAL UNEQUAL A B C)

(EQUIVASSERT EQUAL UNEQUAL D E)
```

places the following indicator-property pairs on the corresponding expressions:

Expression	Indicator	Property
A	EQUAL	(SET A B C)
B	EQUAL	(SET A B C)
C	EQUAL	(SET A B C)
D	EQUAL	(SET D E)
E	EQUAL	(SET D E)

Thus each expression has a set of all the expressions asserted equal to it stored on its property list. Logically the two statements mean A=B=C and D=E.

If we declare other combinations of the original expressions to be unequal, the information is stored as a set of unequal sets. For example, the statements

```
(EQUIVDENY EQUAL UNEQUAL F G H)

(EQUIVDENY EQUAL UNEQUAL H I)
```

place the following indicator-property pairs on the corresponding expressions:

Expression	Indicator	Property
F	UNEQUAL	(SET (SET G H))
G	UNEQUAL	(SET (SET F H))
H	UNEQUAL	(SET (SET F G) (SET I))
I	UNEQUAL	(SET (SET H)) .

Each set of sets represents a conjunct of disjuncts in the following way. For some expressions, say _e_, call its set of sets NES(_e_). Next, name the elements of NES(_e_):NES(_e_) (_i_). Each NES(_e_) (_i_) is a set of expressions, say NES(_e_) (_i_) = (SET _u1_ _u2_ ... _un_). Then NES(_e_) (_i_) represents the disjunct

$$\underline{e}\neq\underline{u1} \text{ or } \underline{e}\neq\underline{u2} \text{ or } \underline{e}\neq\underline{un} \quad .$$

NES(_e_) represents the conjunct of the individual disjuncts. For example, for H in the above table,

NES(H)	is	(SET (SET F G) (SET I))
NES(H)(1)	is	(SET F G)
NES(H)(2)	is	(SET I) .

This property logically represents the assertion

$$(H\neq F \text{ or } H\neq G) \text{ and } H\neq I \quad .$$

3. EQUIVASSERT

Now let us examine the operation of EQUIVASSERT statements, say

(EQUIVASSERT EQUAL UNEQUAL e1 e2 ... en) .

a. Equality Partition

The first step is to form a set, say S, of all expressions relevant to this statement that are known to be equal. To do this we first define, for each ei in the EQUIVASSERT statement, a set ES(ei) as:

If ei has an EQUAL property then ES(ei) is that set, otherwise ES(ei) is (SET ei).

The union of all the ES(ei) is the equality partition, call it S.

S = UNION ES(ei) for all ei in the EQUIVASSERT statement.

b. Consistency Checks

The next step is to check the consistency of values already assigned to each element of S. If two expressions have already been assigned values (normal QA4 values like TRUE), and if these values are different, the expressions cannot be equal, so a failure is generated. Moreover, if some of the expressions have been assigned the same value, that value is assigned all the expressions in S.

The second consistency check assures that the new equality does not conflict with all the previous unequal assertions. Since we are asserting that all the expressions in S are equal, all the unequal

sets of each expression in S apply to every other expression in S. To form the set, NS, of all those unequal sets, we first define, for each ei in S, the set NES(ei) as:

> If ei has UNEQUAL set of sets then NES(ei) is that set, otherwise NES(ei) is the empty set.

We can now define NS as

NS = UNION NES(ei) for all ei in S.

NS now has the form of an UNEQUAL set of sets. Name the elements of NS, NES(i). Each NES(i) has either come from an e in the original statement or it has come from some e known to already equal an e in the original statement. If NES(i) is contained in S for any i, then there is a conflict, for we have previously asserted that not all expressions in a set are equal, while we are now attempting to assert that they are all equal (because they are all in S). This condition is checked; if an inconsistency occurs, a failure is generated.

c. Unequal Partitions

Next a new unequal set of sets, NSS, is computed by taking S from each NES(i). That is, NSS is a set of sets, each subset of NSS is called NSS(i), and each NSS(i) is computed by:

NSS(i) = NES(i) - S .

Remember that each NES(i) represented a disjunct of the form

e≠u1 or e≠u2 ... or e≠un .

Since we are now asserting something of the form

$\underline{e=uj}$ and $\underline{e=uk}$

we may delete uj and uk from NES(i).

If both checks pass, S is made the EQUAL partition for every element of S, and NSS is made the UNEQUAL set for every element of S. If any S had a value, moreover, that value is assigned to every element of S.

d. EQUAL Example

Let us now carry out an EQUIVASSERT with our previous example. Suppose we made the statement

(EQUIVASSERT EQUAL UNEQUAL D F H) .

First we compute

ES(D) = (SET D E)

ES(F) = (SET F)

ES(H) = (SET H)

S = (SET D E F H) .

Next we perform the value check, and let us assume it passes. For the second consistency check, we first compute

NES(D) = (SET)

NES(E) = (SET)

NES(F) = (SET (SET G H))

NES(H) = (SET (SET F G) (SET I))

NS = (SET (SET G H) (SET F G) (SET I)) .

Since none of (SET G H), (SET F G), or (SET I) is contained in (SET D E F H), this check passes. Finally we are ready to compute NSS:

```
NSS = (SET (DIFFERENCE (SET G H) (SET D E F H))
           (DIFFERENCE (SET F G) (SET D E F H))
           (DIFFERENCE (SET I)   (SET D E F H)))
    = (SET (SET G) (SET I))   .
```

Note that empty sets are discarded. To complete the statement, S is added as the equality partition to each element of S, and NSS is added to each element of S as the unequal set of sets. We now have:

Expression	Indicator	Property
A	EQUAL	(SET A B C)
B	EQUAL	(SET A B C)
C	EQUAL	(SET A B C)
D	EQUAL	(SET D E F H)
	UNEQUAL	(SET (SET G) (SET I))
E	EQUAL	(SET D E F H)
	UNEQUAL	(SET (SET G) (SET I))
F	EQUAL	(SET D E F H)
	UNEQUAL	(SET (SET G) (SET I))
G	UNEQUAL	(SET (SET F H))
H	EQUAL	(SET D E F H)
	UNEQUAL	(SET (SET G) (SET I))
I	UNEQUAL	(SET (SET H)) .

4. <u>EQUIVDENY</u>

When we make an EQUIVDENY statement, a completely different operation occurs. Suppose we state

(EQUIVDENY EQUAL UNEQUAL <u>u1</u> <u>u2</u> ... <u>un</u>) .

a. Equal Partitions

The system begins by first forming a set T of all the *ui* and every expression known equal to them. That is,

T = UNION ES(*ui*) for all *ui* in the EQUIVDENY statement.

b. Consistency Checks

The only consistency check is to verify that for each *ui* in the original statement, ES(*ui*) is not identical to T. ES(*ui*), remember, represents the conjunct

ui=*ui1* and *ui*=*ui2* and ...

while T represents the disjunct

ui≠*u1* or *ui*≠*u2* or ... .

Thus ES(*ui*) claims everything in a set is equal while T claims at least two expressions in a set are unequal. Moreover, T is a union of ES(*ui*), so each ES(*ui*) must be contained in T. If any ES(*ui*) is equal to T, we have a contradiction--a set of equal expressions with the claim that two are not equal.

c. Unequal Partitions

For the EQUIVDENY statement we must compute a new set of sets, NSS, for each element of T. Each NSS will be different. For some element of T, say *ej*, the computation proceeds by first computing a set X:

X(*ej*) = T - ES(*ej*) .

Since ES(_ej_) is the set of all expressions equal to _ej_, X(_ej_) is now a minimal set of unequal expressions. Remember that NES(_ej_) is the set of unequal sets already on _ej_. Now to form the new conjunct, we include a reduced form of each element of NES(_ej_) and X(_ej_). To get the reduced sets let us name the sets of NES(_ej_): NES(_ej_)(1), NES(_ej_)(2), etc. Both NES(_ej_)(1) and X(_ej_) represent disjoints; thus, if either strictly contains the other, the smaller subsumes the larger and we may include, as the reduced form, the set difference of the larger less the smaller. If, however, they are partially disjoint, then we could include the union, but since X(_ej_) is included already, we merely include NES(_ej_)(i) itself.

d. UNEQUAL Example

Let us complete our example with the statement

(EQUIVDENY EQUAL UNEQUAL A F I) .

First we compute

ES(A) = (SET A B C)

ES(F) = (SET D E F H)

ES(I) = (SET I)

T = (SET A B C D E F H I) .

The consistency check certifies that (SET A B C), (SET D E F H) and (SET I) are all strictly contained in T. To compute NSS for A we first compute

X(A) = T - (SET A B C) = (SET D E F H I) .

Since no member of A had an UNEQUAL property, the new NSS is (SET (SET D E F H I)). The computations for B and C are the same. For D

X(D) = T - (SET D E F H) = (SET A B C I)

NES(D) = (SET (SET G) (SET I)) .

The reduced form of (SET G) is just (SET G). (SET I) however, is strictly contained in X(D) so its reduced form is (SET A B C). Thus the NSS for D is

(SET (SET G) (SET A B C) (SET A B C I)) .

The computations for E are the same and the computations for F and I quite similar.

D. Special Context and Recommendation Options

A ctx-rec may occur on any of the property manipulation statements. Some, however, are relevant only to statements that change properties (GET only peeks at the property). All except the context option are merely key words without parameters that are listed at the end of the statement. These options may be classified into four groups.

1. Special Context

The form WRT v, where v is a variable that has been bound to a context (see the Section on Context Statements), will force the statement to use the requested context rather than the standard most-global-dynamic current-backtracking. That is, this statement will operate with respect to v. This option applies to all the statements.

2. *Changing Properties

a. CHANGES

This permits the statement to change a property. The normal case is assumed to be one where properties are not changed once they are set. Thus, if the same property is reassigned nothing happens. However, if an attempt is made to change the property without the CHANGES option, a failure is generated.

b. FAIL-IF-NO-ACTION

This not only permits a change, but causes a failure if one does not occur. Neither of these options apply to EQUIVASSERT and EQUIVDENY statements.

3. *Demon Control

The key word NO-WHENS forces the statement to bypass the WHEN checks. That is, if this option is present, no WHEN programs will be activated. This option applies to all the statements.

4. *Backtracking Control

a. PERMANENT

This forces any changes to remain even if a failure backtracks over this statement. This is accomplished merely by using the second backtracking element of the backtracking context as the backtracking head and moving the property if it is currently stored under the present backtracking head.

b. TEMPORARY

This forces the change to be removed if the current sub-program successfully returns to its calling program. This is accomplished by using the current dynamic context rather than the most global. Both apply to all the statements except GET.

VI QUERY STATEMENTS

A. Motivation

As QA4 programs run, they construct and modify a global data base. The items in this data base are organized as properties of expressions. The data base is indexed with patterns. For example, the GET and PUT statements perform property manipulation. The statements, however, use the pattern in an explicit way:

```
(GET (P A) PROP1)
```

fetches the value of PROP1 from (P A) while

```
(GET (P ←X) PROP1)
```

fetches the value of PROP1 from (P ←X). Get does not permit the program to ask: Find all expressions that match (P ←X) and have PROP1.

B. INSTANCES

1. Example

The INSTANCES statement permits QA4 programs to associatively, according to patterns, query the global data base. If

```
(ASSERT (P A))
(ASSERT (P B))
```

had been executed, the statement

```
(INSTANCES (P ←X))
```

would evaluate to

```
(SET (P A) (P B))    ,
```

the set of expressions that match the pattern and are TRUE.

2. Form

The full form of the statement is

(INSTANCES pexp c-r)

where pexp is a pattern and c-r is a list of context modifications and restrictions. If c-r is absent, the expressions found must have the value TRUE. This is commonly the only necessary restriction. Normally, some restriction is necessary for there may be many irrelevant expressions in the data base that match pexp. For example, (P ←Y), (P $X), and (P (SET 1 2)) all match (P ←X). These may, however, be in the data base merely because they exist as forms in other programs. If c-r is absent, the context under which the expression must have value TRUE is the global-dynamic current-backtracking. This is the same default as the property manipulation statements.

3. Restrictions

c-r can be a list of property-value pairs and context modifications. If MODELVALUE is one of the properties, then the value indicated is used for the check. If MODELVALUE is not listed as a property, then is must have value TRUE. To avoid checking it, the pair IGNORE NETVALUE must be included in c-r. Just as in Property Manipulation statements WRT v alters the context for succeeding checks to v. For example

(INSTANCES (P ←X) WRT $C P1 V1)

will require expressions that match and have a property P1, with value

V1 in the context $C . Alternatively, for all e in the set

(INSTANCES (P ←X) WRT $C P1 V1)

we have that

(GET e P1 WRT $C) = V1 .

3. Evaluation

The INSTANCES statement operates by first instantiating pexp to, say pexpi. Then the data base is searched and a list made of all expressions that may match pexpi. This heuristic search is described in Appendix II, The Discrimination Net.

Next, for each expression c-r is checked, and for those that pass, the pattern matcher is used to verify that they match pexpi. The value of the statement is the set of expressions that pass both tests.

C. EXISTS

The EXISTS statement also retrieves expressions from the global data base and uses the same c-r. However, it is coupled with the back-tracking mechanism and appears to retrieve the expressions one at a time. The form is

(EXISTS pexp c-r)

where pexp and c-r are the same as in the INSTANCES statement. The interpreter first instantiates pexp to pexpi and searches the data base for potential matching expressions. The chosen expressions are examined one at a time until one is found that satisfies c-r and matches pexpi. A backtracking point is established and the ← variables

of *pexpi* are bound to the appropriate parts of the selected expression. If a failure causes backtracking to this point, the examination is resumed. When the chosen expressions are exhausted, a failure is generated.

For example,

```
(EXISTS (NEXTTO (BAG ROBOT ←X)))
```

may first find

```
(NEXTTO (BAG (ROBOT BLOCK1)))
```

and bind X to BLOCK1. After a failure to this point another NEXTTO expression may satisfy *c-r* and the program may resume with X bound to something else. Further failures would eventually cause the statement to fail.

The value of the EXISTS statement is the current chosen expression. Thus

```
(SETQ ←Y (EXISTS (NEXTTO (BAG ROBOT ←X))))
```

would bind Y to (NEXTTO (BAG ROBOT BLOCK1)) as well as bind X to BLOCK1.

VII CONTEXTS

A. Canonical Representations

Every QA4 expression is always constructed in a canonical form. This process is discussed in II-A, Types and Formats and in Appendix II, The Discrimination Net. Together with the query statement, this provides a rich associative indexing scheme into the total data base of the system. In QA4 only one instance of an expression may occur, thus all the expressions of the language may be used as data stores and as communication paths between running processes. In LISP, ATOMS have property lists that are often used to store data. This is possible in LISP because every instance of the same ATOM is a pointer to the same word in memory, and that word is the head of the property list. In QA4 every expression is a pointer to a word that is the property list for the expression. The syntactic form of the expression is only one of the properties of the expression. This property, known as EXPV, corresponds to the PNAME of a LISP ATOM.

B. Bindings of Properties

As another extension of LISP in a similar vein, QA4 extends the notion of variable bindings to the properties of expressions. Within LISP, property list modifications are permanent and never restored to previous states automatically. This could be thought of as a "top level" binding. Since top level always remains at the top, the binding mechanisms of LISP never manipulate property lists. (There is a slight

exception in the way that some LISP systems handle SPECIAL variables, but this is unnecessary.)

In QA4 all statements that manipulate properties may use either the top level, or the current level, or any level they wish. That is, the WRT option on statements such as PUT and SETQ permit the user to specify precisely which context is to be used. A context level is created every time a function application is performed or a PROG is entered. When the PROG or function is exited, the level is popped and everything modified with respect to that level is restored to its state before the level was created.

The defaults of the QA4 system give the same effect as LISP. The SETQ statement uses local context when it changes the value property of a variable. All the other statements change values of properties at the top level or the most global context. Since the only data base available to QA4 programs is QA4 expressions and their properties, all data base changes are made under a binding or context mechanism.

The extension of binding to all data base changes compensates for the generality of the canonical representations. When multiple processes are operating, each process will require that its changes be local to it. That is, suppose process P begins two subprocesses P1 and P2, and runs them as co-routines. P1 and P2 each have a different context assigned at the time of their creation that remains throughout their lives as processes. And each may manipulate properties

with respect to P's context without interfering with still other processes within the system. In effect, top level has become a relative notion, and groups of programs organized around process-structures may use a relative top level for their group.

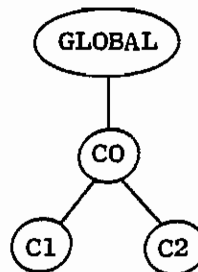
C. User Contexts

QA4 was specifically designed to aid the construction of programs that perform robot planning and theorem proving. These two areas of artificial intelligence must constantly deal with a binding problem similar to the variable binding problem for programming languages. For robot planners it is called the frame problem (McCarthy and Hayes, 1969). Simply, the model of the robot world is large and individual actions modify the model only slightly. Thus the planner needs a mechanism for changing the model, and after exploring the effects of the change, restoring the model to a previous state. For theorem provers, the problem is one of hypothetical reasoning. The theorem prover must make an assertion (which is really an assumption), carry out a proof, and then remove all the side effects of the proof as well as the assumption. To prove (A implies B) for example, we would assume A, prove B, and remove A and all the proof steps, and then assert (A implies B).

Both these problems share many similarities with the regular variable binding task for programming languages. It is inconvenient, however, to attempt to solve the user problems by forcing the robot planner or theorem prover to use the standard variable binding mechanism

to control the robot model or theorem base. Instead, an independent binding mechanism can be made available that is controlled by the user program but not directly tied to the particular way his programs call one another (McDermott, 1972). For example, a robot planner may have an initial model in the global context and derive a context CO from that global context.

It could move the robot in CO and explore the consequences. It may then find two alternatives to explore and create two processes P1 and P2, but it could now derive two new, independent contexts C1 and C2 from CO.



There may have been function calls and contexts created, yet the planner is manipulating a model that reflects only the changes relevant to the actions of the robot. When a path is no longer relevant, the context may be popped. Moreover, since the tree of contexts does not contain irrelevant nodes, it is easy for the programs to explore values in various parts of the tree.

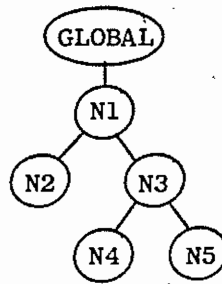
D. CONTEXT Statement

The context statement has two forms

(CONTEXT PUSH c)

(CONTEXT POP c)

where c is either GLOBAL, CURRENT, or a variable that is bound to a context. The value of the statement is a new context. For example, the tree



could be grown by executing the following series of statements:

```
(SETQ ←N1 (CONTEXT PUSH GLOBAL))

(SETQ ←N2 (CONTEXT PUSH $N1))

(SETQ ←N3 (CONTEXT PUSH $N2))

(SETQ ←N4 (CONTEXT PUSH $N3))

(SETQ ←X (CONTEXT POP $N4))

(SETQ ←N5 (CONTEXT PUSH $X))    .
```

Note that N3 and X are the same context.

E. Summary

QA4 uses one general mechanism for all context manipulation and property retrieval. This mechanism is based on a "dispersed state"

rather than the more common stack mechanisms. The programs and their operations are described in Appendix III. This unusual method was chosen for QA4 because of the experimental nature of the language. The language was designed to permit research into process structures and their relation to backtracking. Neither of their programming concepts have been used extensively enough to freeze a design within a language. With QA4 we have to evolve a semantics by constructing robot planners and theorem provers. During this evolution, the intricacies of both processes and backtracking will most probably undergo radical modification. The context mechanism will, we hope, permit this evolution with minimum effort.

VIII STANDARD CONTROL STATEMENTS

A. LIST Statement

Many of the QA4 statements permit a list of statements as part of their form. In the user form of the statements the list is enumerated, but the preprocessor gathers up the statements of the series and constructs a single LIST statement from them. This LIST statement may also be used any place a single expression is required in a form. The form of the LIST statement is

(LIST e1 e2 ... en)

where an e is an arbitrary QA4 expression. The internal form is

(STATEMENT LIST e1 e2 ... en) .

The LIST statement is evaluated by evaluating e1 through en in order. The value of the LIST statement is en.

In order to simplify the description of other statements, we will call a series of statements that are automatically converted to a LIST statement by the preprocessor a list-segment.

B. Conditionals

1. IF Statement

The IF statement has the user form

(IF e1 THEN e2 ELSE e3)

where e1, e2 and e3 are list-segments. That is, they may be a single statement or a series of statements. If they are a series, they will be gathered together by the preprocessor and converted to a LIST

statement for the internal form of the IF statement. Both the THEN and ELSE parts of the statement are optional. That is, either or both and their corresponding list-segments may be absent. The value of the IF statement is:

If the value of _e1_ is FALSE

then (if there is an ELSE part then the value

of _e3_ else the value of _e1_ (which is FALSE))

else (if there is a THEN part then the value of _e2_

else the value of _e1_) .

2. ATTEMPT Statement

The ATTEMPT statement provides a way that the program may be protected from unexpected failures. The intent of the statement is to provide a barrier that can catch failures. The form of the statement is

(ATTEMPT _e1_ THEN _e2_ ELSE _e3_)

where _e1_, _e2_, and _e3_ are list-segments. To understand the way ATTEMPT is evaluated, we must distinguish between a failure during the evaluation of _e1_ and failure for the completed evaluation of _e1_. _e1_ may contain many backtracking points, especially if it has function calls. Thus, during the evaluation of _e1_ there may be many failures, yet the evaluation may eventually be successful. However, if the failures exhaust the alternatives of the first backtracking point, then the evaluation fails. Normally, the interpreter would backtrack to the choice point established prior to the evaluation of the LIST statement

(list-segment el). The ATTEMPT statement erects a barrier to the failure mechanism so that a failure for the evaluation of el transfers control to the ELSE part of the ATTEMPT statement rather than a prior backtracking point. Just as in the IF statement, the THEN and ELSE parts are both optional. Thus the value of the ATTEMPT statement is:

```
If the evaluation of el fails
     then (if there is an ELSE part
          then (if the evaluation e3 fails
               then FALSE
               else the value of e3)
          else FALSE)
     else (if there is a THEN part
          then (if the evaluation of e2 fails
               then FALSE
               else the value of e2)
          else FALSE).
```

From this we can see that the ATTEMPT statement cannot fail. It is the only statement of the language that has this property. Another feature of the statement is that, once it has been executed, it cannot be failed back into. That is, once the statement has been executed, all backtracking points that may have been established are removed. Thus the barrier to failures works both ways. A program can be protected from an automatic restart due to failure as well as inhibit

failures from causing restarts outside the ATTEMPT. Finally, no part of the statement may fail into another. That is, even if el contains backtracking points, and is successfully evaluated, the evaluation of e2 cannot fail back into el.

3. CASES Statement

a. Form

The CASES statement provides a way of selecting a program to run based on pattern matching. The user form of the statement is

(CASES t pexp) .

t must be a pattern that instantiates to a tuple of lambda expressions, and pexp is a pattern. The statement is evaluated by matching the instantiated form of pexp, say pexpi, against the bound variable part of each lambda expression in turn. When a match is found a backtracking point is established, and the lambda expression is applied to pexpi. The value of the CASES statement is the value of the application. If the application fails, the matching of pexpi against the bound variables continues. If t is exhausted, a failure is generated for the CASES statement.

b. Examples

Suppose we executed the statement

```
(CASES (TUPLE
        (LAMBDA (TUPLE :←X 0) ($UGH :←X))
        (LAMBDA (TUPLE :←X :←Y) (DIVIDES :$X :$Y)))
        (TUPLE $A $B))    .
```

If D is 0 it calls function UGH with the value of A; otherwise it divides A by B. As another example, suppose the variable ACTION is a tuple of function names. We may have executed

```
(SETQQ ←ACTION (TUPLE $FOO1 $FOO2))    .
```

Notice that we used SETQQ. Then the statement

```
(CASES (= $ACTION) (P $X))
```

would attempt to apply the instantiated form of (P $X) to FOO1 and FOO2 in turn.

We could alternatively set ACTION to the tuple of lambda expressions by executing

```
(SETQ ←ACTION (TUPLE $FOO1 $FOO2))
```

and then executed

```
(CASES $ACTION (P $X))    .
```

This can lead to bugs, for if we edited the function FOO1 after we set the value of ACTION, the CASES statement would still use the old form of FOO1. Since this is often the case when variables such as ACTION are part of the initial model and we are interactively debugging the QA4 programs, this second form should not be used.

c. Unbound Variables

Suppose FOO1 were

(LAMBDA (P ←Z) (SETQ ←Z 3))

and we executed

(CASES (= $ACTION) (P ::←X)) .

Then pexpi would be (P :←X) and when it is matched to (P ←Z), Z would match ←X. Thus Z would be bound to ←X. This is, in a sense, treating ←X as a quoted form.

The case where ←X is not considered quoted, however, occurs in QA4 but not in most other programming languages. For example, in

(CASES (= $ACTION) (P ←X))

we cannot bind Z to ←X. Instead, Z is bound to NOSUCHPROPERTY, indicating it has been bound, but not yet to an object. It is in a state of limbo. After FOO1 has been executed, the value of X with respect to the context of the CASES statement is set to the value of Z with respect to the initial application context of FOO1. In our example, after executing

(CASES (= $ACTION) (P ←X))

X would have the value 3.

d. The WRT Option

A CASES or GOAL statement may contain a WRT option with a user-generated context. For example,

(CASES (= $ACTION) (P X) WRT $C)

where $C was set to the value of a CONTEXT statement. When this option is used, a variable, CTXREC, is set to the WRT context, and the SETQ is done in the current dynamic context of the program in which the CASES or GOAL statement occurs. Whenever any property manipulation statement or query statement is executed, this variable is checked. If it has a value in the context in which the ASSERT or EXISTS statement is being executed, then its value is used as the context for the ASSERT or EXISTS. This has the effect of permitting a GOAL statement to establish a subcontext. If all the property manipulation statements in the program that work on the goal do not use a WRT clause, then all the model manipulation is performed in the appropriate subcontext.

C. Programs

The PROG, RETURN, and GO statements play a role similar to their counterparts in LISP.

1. PROG

The form of the PROG statement is

(PROG (DECLARE *v1* *v2* ... *vn*) *e1* *e2* ... *em*)

where each *v* is a variable and each *e* is either a label or an expression. Each *v* is an unprefixed variable and is initially bound to NOSUCHPROPERTY to indicate that it does not yet have a value. If an *e* is an identifier, that is an unprefixed atom, it is treated as a label; otherwise it is treated as a statement of the PROG. The PROG

is evaluated by first creating a context and binding each _v_. Then each _e_ is evaluated in turn. Labels are not evaluated, and GO statements cause control to transfer to the appropriate label. The value of the PROG is the value of the last statement executed; if no RETURN statement is executed, the value of the PROG will be the value of _em_.

2. GO

The form of the GO statement is

(GO _v_)

where _v_ is an unprefixed identifier.

3. RETURN

The form of the RETURN statement is

(RETURN _pexp_) .

The RETURN statement causes the PROG to be exited. The value of the PROG will be the instantiated form of _pexp_.

4. Example

The following function computes the factorial of its argument.

```
(LAMBDA ←X
    (PROG (DECLARE Y)
          (SETQ ←Y 1)
    LOOP (IF (LTQ $X 1)
              THEN (RETURN $Y)
              ELSE (SETQ ←Y (TIMES $X $Y))
                   (SETQ ←X (SUBTRACT $X 1))
                   (GO LOOP))))   .
```

D. Failure

1. FAILING? Statement

The form of this statement is

(FAILING? exp)

where exp is any QA4 expression. The statement has the effect of establishing a backtracking point with exp as the alternative. That is, exp is not evaluated when the FAILING? statement is executed. Instead, a backtracking point is established. If failures eventually cause backtracking to this point, exp is then executed.

As an example, suppose we have in our model a number of expressions of the form (P ←X), say (P 1), (P 2), etc. Also suppose we wish to try a Plan-A with one of the forms but we are not certain which one. Plan-A will fail if the chosen one is not correct. If none of the expressions work for Plan-A, we then wish to try Plan-B. The program could take the following form:

```
(FAILING? Plan-B)
(EXISTS (P ←X))
Plan-A .
```

Notice the difference of this program with

```
(ATTEMPT (EXISTS (P ←X)) THEN Plan-A ELSE Plan-B) .
```

With the ATTEMPT statement only the first choice of (P ←X) is tried for Plan-A. If it does not work and Plan-A fails, the ATTEMPT is immediately

finished with value FALSE. Only if no expression of the form (P ←X) exists will Plan-B be tried.

2. FAIL Statement

The FAIL statement is a way of forcing a failure. Its form is

(FAIL) .

It causes a failure to be generated.

IX WHEN STATEMENTS

A. Motivation

Problem-solving programs written in QA4 are organized around strategies such as

> Whenever a sentence follows from previous assertions by modus ponens, assert that sentence.

Heuristics such as these are easily introduced into a problem-solving system by using a WHEN statement. The statement directs the interpreter that whenever any expression that matches a particular pattern is assigned a property,execute a program.

WHEN statements correspond to "demons." Conceputally, one can think of demons that watch the flow of data between programs. When data that triggers a demon moves past its watching post, it executes a program causing side effects. In a speech understanding system, for example, a user might demand "When you remove a triangle also remove a circle." A QA4 problem solver could easily accommodate these directions by first synthesizing a program that deletes a circle. A demon could then be generated by a WHEN statement that monitored the REMOVE program and activated the circle deletion program when that demon notices a triangle being removed. This simple method of carrying out and understanding is particularly awkward outside the framework of a system such as QA4, yet simple and natural within the QA4 language.

QA4 problem solvers evolve through interactive programming. At first the programmer guides strategies to their solutions. As insights develop, the strategies are tuned. Developing the insights, however, requires extensive trace features. The WHEN statement is such a full trace system.

B. Form

1. Overall Form

The WHEN statement takes the form

(WHEN trigger-condition THEN event) .

trigger-condition refers to a dynamic situation. These situations occur when a process sends a message WHEN or a variable is bound to a value. The event is the statement that is executed whenever the condition occurs. These usually take the form of a PROG statement or a function call. The demon is a process that assumes the global dynamic context unless a more restrictive context is specified. The process is directly assimilated into the interpreter and activates the event as a subprocess whenever the condition occurs.

2. trigger-condition

The form of the trigger-condition is item-to-monitor direction pattern wrt.

For example,

(WHEN X RECEIVES 5 THEN ($FOO $X))

specifies that whenever the variable X receives the value 5 the program

FOO is to be applied to X. The <u>item-to-monitor</u> is X, the <u>direction</u> is RECEIVES, the <u>pattern</u> is 5, and the <u>event</u> is ($FOO $X). <u>wrt</u> could restrict the scope of the monitoring to the usage of X within a specific process.

The WHEN mechanism can be viewed as a tribe of demons. Each demon is assigned either a collection of expressions or a process. He watches the data path of his <u>item-to-monitor</u>. What he sees are all expressions that go in <u>direction</u> (to or from the item) past his watching post. He is, however, awake only when the processes in context <u>wrt</u> are active. When he notices an expression that matches <u>pattern</u> he causes <u>event</u> to become the active process. When <u>event</u> is finished, he reestablishes the process he suspended.

a. <u>item-to-monitor</u>

The item may be either a class of expressions or a process. The most simple class of expressions is a variable. Larger classes are specified by using a pattern.

- <u>Expressions and Patterns</u>

Expressions have the most extensive set of specifications. Merely to give the name of a variable may result in an enormous number of unnecessary invocations of the testing procedure. There is, therefore, an option that restricts the domain of testing. The form of <u>item-to-monitor</u> for patterns is

EXP <u>pexp</u> INDICATOR <u>ind</u> .

Only EXP pexp is necessary for it specifies exactly which expressions to monitor. If pexp is a simple variable, say X, then just that variable is monitored. pexp may be, however, any complex pattern expression. For example,

EXP (AT ROBOT ←X)

might monitor all instances of the AT predicate involving ROBOT. The item

EXP (←F ←X 7)

on the other hand, monitors all two-argument predicates that have 7 as their second argument.

The INDICATOR option permits the WHEN test to be restricted to an individual indicator or property of the expression. For example:

INDICATOR NETVALUE

would restrict the monitoring to only the value of the expression. However, a program may be using many other properties. For example:

INDICATOR TIMEARRIVED

might be the real time the AT used above was established. The full specification

EXP (AT ROBOT ←X) INDICATOR TIMEARRIVED

would monitor the AT predicate about the ROBOT, but will only watch the properties associated with TIMEARRIVED rather than NETVALUE. In this case the predicate may have the value TRUE, an uninteresting

aspect of the expression. The useful information is the location of the ROBOT, which would be bound to X, and the time, that can be bound to variables in the pattern portion of the trigger condition. The default option for INDICATOR is NETVALUE.

- Processes

Processes also have a simple item-to-monitor. Its form is

PROCESS e

where e instantiates to an expression of type process. The form specifies that all RESUMES into or out of the process e are to be monitored. e is usually a $ variable set to the value of an INCARNATE statement.

a. direction

This is yet another way of restricting the scope of the demon. direction is one of the three words: SENDS, RECEIVES , or ERASES. In the case of a process, it SENDS when it executes a RESUME statement and it RECEIVES when another process sends it an expression. ERASES does not apply to processes. An expression SENDS when a program reads a property from an indicator. Expressions RECEIVE when the property for an indicator is set as the result of a PUT or SETQ statement. ERASES applies only to expressions and is triggered by the ERASE statement. The following table shows the actions that are related to direction.

direction	expression	process
SENDS	GET	RESUME
RECEIVES	PUT	activated
ERASES	ERASE	does not apply

The default for direction is to watch all the activity of the item.

b. pattern

The pattern both restricts the cases that activate the event and provides a way of binding variables global to the event to the parts of the message that are of concern. Suppose we wish to monitor all predicates of two arguments and cause an event whenever one becomes TRUE. The statement

```
(WHEN EXP (←F ←X ←Y) RECEIVES TRUE ...)
```

used the simple pattern TRUE to accomplish the task. As another example, we may use a tuple pattern as the TIMEARRIVED property of the AT predicate. Then

```
(WHEN EXP (AT ROBOT ←X) INDICATOR TIMEARRIVED
    RECEIVES (TUPLE ←HOUR ←MINUTE ←SECOND) ...)
```

will cause the event when ROBOT moves, bind X to the new location, and bind HOUR MINUTE and SECOND to the time.

c. wrt

The final restriction enables each particular WHEN to have relevance only witn respect to a certain context. The form of

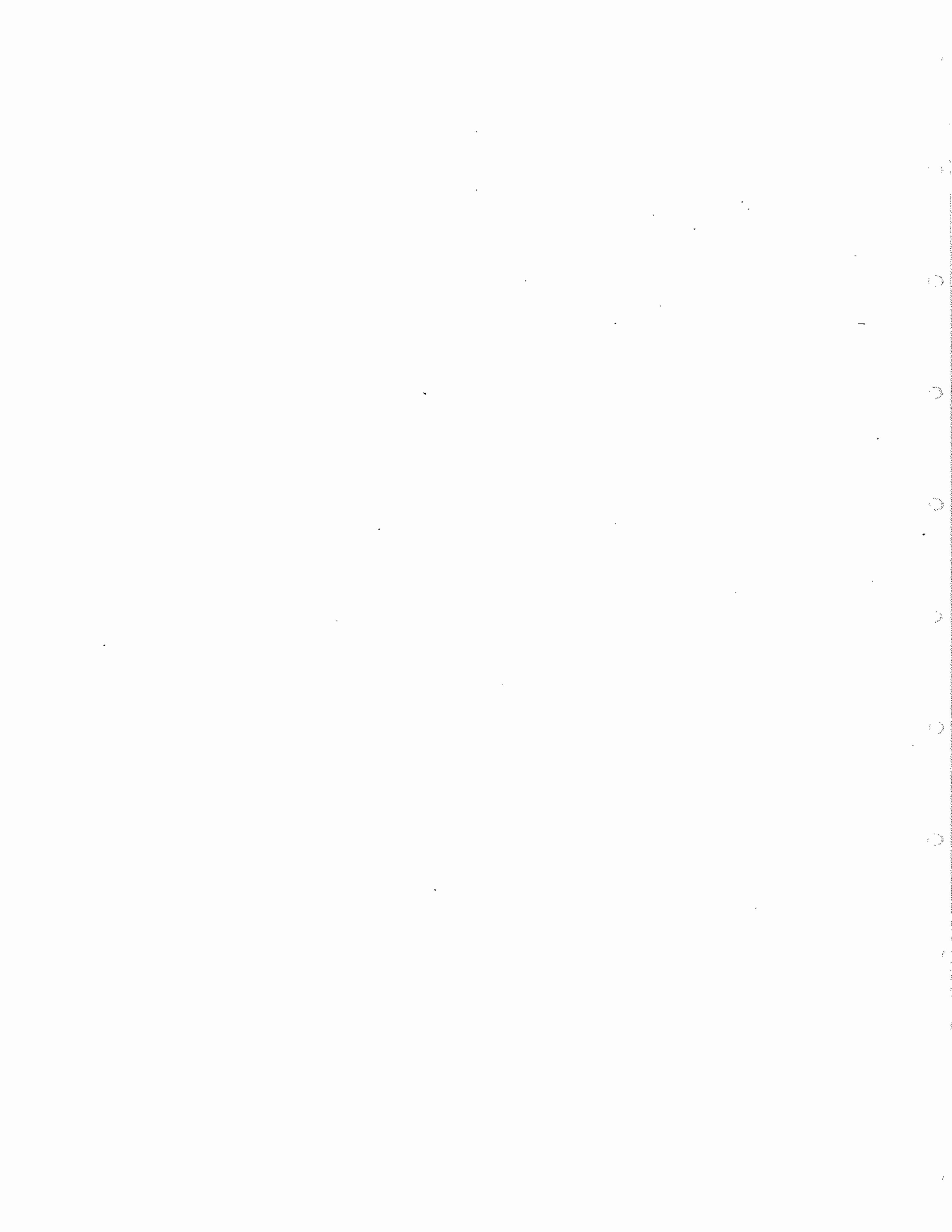
the *wrt* option is:

WRT *context* .

context is an expression that must instantiate to a context. This context can narrow the scope of the monitoring of the item to times when *context* is an active *context*. The WRT option is especially useful for large special-purpose problem solvers. Suppose a problem solver had a linear-equation solver and used it as a high-level inference rule. Part of this subproblem strategy might be to monitor certain variables. Each time it was entered, this subproblem could execute WHEN statements with the appropriate WRT options. Even if the system was suspended, and later resumed, the expressions would not be monitored during the execution of the intervening processes.

3. *event*

event must be a single expression. It may be a PROG or as simple as a function call. When the demon is triggered a context is created for *event*. The variables in the *trigger condition* that are bound as a result of the pattern matching are bound in this newly created context. *event* is then evaluated.



X GOAL STATEMENTS

A. Form

GOAL statements provide a powerful tool for program organization. Their form, however, is quite simple.

(GOAL t pexp c-r)

where pexp is a pattern, t is an expression that instantiates to a GOALCLASS tuple, and c-r is the context-restriction option of the query statements. The execution of the GOAL statement can be defined in terms of more primitive statements:

(FAILING? (CASES t pexp))

(EXISTS pexp c-r) .

B. Example

Normally, the flow of control of a program is completely determined by the code literally appearing in the form of subroutine calls and transfer statements. These two control mechanisms, taken in isolation, are a formidable barrier to the organization and structuring of programs that are mainly guided by the construction and satisfaction of subgoals. A theorem prover, for example, should operate by accepting and analyzing its input; making assertions about it; developing subgoals; and activating programs especially suited to prove each subgoal. In this organizational scheme, programs are not identified by their names but rather by the job they do and the type of input they operate upon.

Suppose, for example, the system has somehow obtained these two facts:

fact1 (ON BLUEBLK REDBLK)

fact2 (ON REDBLK GREENBLK) .

By calling these facts, we mean that their value is TRUE. Now suppose our program has developed a subgoal. It is necessary to find a block that is on the green block. The statement

(GOAL $PROVE (ON ←X GREENBLK))

would accomplish the subgoal.

The first step in the interpretation of the GOAL statement is to make a list, say L, of all expressions in the system that could match the pattern (ON ←X GREENBLK) and that also satisfy c-r. Since c-r is absent, the expression must have property TRUE under the indicator NETVALUE. In our example only one expression is on L, namely fact2. The system chooses an expression from L, binds X to the appropriate subpart, establishes a backtracking point, and continues. When a failure occurs, the next item from L is chosen. This process would continue down the list until one was found that did satisfy the goal and permitted the program to come to a successful termination or until the list was exhausted.

To carry our example a step further, suppose we want to extend ON to mean "directly on or somewhere in a pile on." We could introduce a program that attempts to satisfy ON goals. First we would define a

recursive function that decides if one block is on another. We call it PILE and give it the definition

```
(LAMBDA (ON ←B1 ←B2)
     (PROG (DECLARE B3)
          (GOAL $PROVE (ON ←B3 $B2))
          (GOAL $PROVE (ON ←B1 $B3)))   .
```

Next we link the program PILE to the GOAL statement by executing

```
(SETQ ←PROVE (CONS $PILE $PROVE))   .
```

This means that the system can use the program PILE to satisfy the GOAL statement provided that the goal matches the bound variable of PILE.

Now, if we develop the goal

```
(GOAL $PROVE (ON BLUEBLK GREENBLK))   ,
```

the system would first attempt to construct a list of expressions that might satisfy the GOAL. This fails, for the list is empty, so the system next finds programs from the tuple $PROVE whose bound variables match the goal. In our case, the function PILE would be located and applied to the goal pattern. B1 would be bound to BLUEBLK and B2 would be executed like our first example, and the goal would be satisfied when PILE concludes its execution.

C. Order and Advice

The order in which programs are attempted is taken directly from *t*, the GOALCLASS tuple of the goal statement. Thus the program can, itself, change the order by manipulating the values of the variables used in the GOALCLASS. This is not only a way of guiding the search to more direct solutions, but of being sure that excessive amounts of time are not wasted continually rejecting some programs. It may also serve as a method of permitting the system to advise itself of better search methods. After a problem has been solved, the problem solver would record the successful programs for various kinds of goals. When a similar problem arises, variables could be set to tuples arranged so that the successful programs are at the heads. In this way the problem solver would go directly to the solution of this second problem.

XI PROCESSES

A. RESUME Statement

1. Example

A process is an object that can be executed. It is made up of a context, a program, and state information (Conway, 1963) (Landin, 1963). Any process can be suspended at arbitrary points and restarted at any later time. When the processes work in cooperation, however, they are suspended and restarted only during the evaluation of certain statements. Suppose, for example, that we have two cooperating processes: A and B. They communicate in two ways: through a global data base and by their RESUME statements. The following figure is a flow-of-control diagram that illustrates the interplay of the two processes. Underneath each process name is a list of the RESUME statements for that process in the order they are executed. The annotation describes how these suspension points are reached and how each process is eventually restarted.

Control Between RESUME statements

```
A                        B  ........................ T1
.
.
(RESUME $B 1)            .  ........................ T2
                         .
                         .
                         (RESUME $A 2)  ............ T3
.
.
.
(RESUME $B 3)            .  ........................ T4
```

There are four distinguished points of time T1 through T4.

T1 • A has started execution. We will see in a later section how it is started.

• B is a suspended process. Its state requires that an argument for Bs LAMBDA expression definition be supplied so that the first step in its execution may take place, namely the binding of its argument. This argument will be supplied the first time B is resumed.

• Suppose that B is an expression of the form

(LAMBDA ←BX ...) .

T2 • A executes a RESUME statement. This particular one sends the message 1 to B.

• A now becomes a suspended process. Its state is left in such a way that it requires a value for the RESUME statement in order to be restarted and continue execution.

• The message is sent to B. It becomes the argument to B and BX is bound to 1.

• B then proceeds in execution.

T3 • B executes a RESUME statement. This time the message 2 is sent to A.

• B now becomes a suspended process requiring a value in order to continue.

- The message is received by A, and 2 becomes the value A's RESUME statement, and A continues execution. Thus each RESUME statement serves two purposes: it sends a message and receives a message in return.

T4 • A executes a RESUME statement and sends a 3 to B.

- A becomes suspended and B begins execution with 3 as the value of its RESUME statement.

2. <u>Form</u>

The form of the RESUME statement is

(RESUME <u>process</u> <u>message</u>)

where <u>process</u> instantiates to a QA4 process and <u>message</u> is the expression that is instantiated and sent to the process. When a RESUME statement is executed, its process is suspended. The suspension is made in such a way that a value is required as the value of the RESUME statement before the process can be restarted. This value can be supplied by a RESUME statement from any other process in the system. Normally, <u>process</u> is a variable that is the value of an INCARNATE statement. Thus, processes are linked together merely because they have variables that point to other processes. This means that there is no rigid structure to QA4 processes as there often is in systems that require pipes or plugs and sockets. Moreover, the interconnections can be dynamic and continually change within a QA4 process structure.

B. INCARNATE Statement

Suppose that F and G are functions. The following lines of program may have created the above example.

```
(SETQ ←A (INCARNATE $F))

(SETQ ←B (INCARNATE $G))

(RESUME $A 0)    .
```

The form of the INCARNATE statement is

(INCARNATE _e_)

where _e_ instantiates to a lambda expression. The value of the INCARNATE statement is a process that needs a value to begin execution. The lambda expression indicated by _e_ will be applied to the message from the first RESUME statement that resumes the process. For example, suppose F has the definition

```
(LAMBDA (Q ←X)  ..)
```

and we execute

```
(SETQ ←P (INCARNATE $F))

(RESUME $P (Q 7))    .
```

_e_ of the INCARNATE statement is $F and this instantiates to the LAMBDA definition of F. The value of the INCARNATE statement is a process, and P is bound to that process. The RESUME statement supplies the _message_ (Q 7) to the process. Since this is the first resume to the process, the lambda expression is applied to the _message_. That is, (Q ←X) is bound to (Q 7) and the process is started.

A particular LAMBDA expression can be incarnated many times. We could, for example, have a program that reads:

```
(SETQ ←C (INCARNATE $F))
(SETQ ←D (INCARNATE $F))    .
```

C and D are different processes with different contexts. They share the same code but will have different variable bindings. Since this multiple incarnation is possible, references to processes must be made via the process and not the function name.

C. CONNECT Statement

The CONNECT statement has the form

(CONNECT p a q b)

where a and b are variables that are bound to processes and p and q are identifiers. The statement assigns p the value of b under a's context, and it assigns q the value of a under b's context. For example, suppose we execute

```
(SETQ ←A (INCARNATE $F))
(SETQ ←B (INCARNATE $G))
(CONNECT P $A Q $B)    .
```

The following diagram shows the situation after the CONNECT statement. Arrows indicate values, circles indicate processes, variables within circles indicate what the value is with respect to the context of the process.

D. <u>WAIT Statement</u>

A newly incarnated process is called the child of the process that incarnated it, and we refer to the creating process as the mother of its child. Many times a child wants to RESUME to its mother, but lacks a way of referring to her. The WAIT statement provides a mechanism for this.

(WAIT <u>message</u>)

operates just like a RESUME to the mother process, but does not require that the mother process be explicitly mentioned.

Once a process has been created it may be passed between functions just like any other data object. It can thus take on a life independent of its mother. This violates ALGOL scope conventions and normal stack operations. When all references to a process are removed, the process automatically disappears. That is, when there are no longer any variables that are bound to a process, the garbage collector will come and get it.

XII ITERATION STATEMENTS

A. REPEAT Statement

1. Example

In many languages, iteration statements are intended to step through indexed data structures: FOR in ALGOL and DO in FORTRAN for example. In QA4, however, they are intended to enhance the utility of the nondeterminism of the language. As a result, they operate through the failure mechanism rather than simple transfers. They provide two features: iteration through data structures or multiple pattern matches without the loss of side effects, and collection of items at each iteration step. For example,

```
(REPEAT (EXISTS (PAND ←Y (ON-TOP-OF ←X BRICK1)))
     DO ($PICKUP $X)
        ($PLACE-IN $X BLUE-BOX))
```

is a statement that puts all the bricks on BRICK1 in the blue box. REPEAT is the major statement, and all the other iteration statements are defined in terms of it. Some of the options of the REPEAT are included to facilitate the definition of the other statements and may not appear useful.

2. Form

The form of the REPEAT statement is

(REPEAT <u>nde</u> <u>options</u> DO <u>e1</u> <u>e2</u> ... <u>en</u>)

where _nde_ is a nondeterministic expression, each _e_ is an expression, and _options_ are pairs of keywords and expressions. The options are:

- A terminal condition and a terminal action
- A collection condition and an item
- An iteration step
- A final failure expression.

These will be discussed more fully below. The execution of the REPEAT may be described by the following pseudo-QA4 program:

```
(SETQ ←COLLECTION (TUPLE))
(FAILING? failure expression (RETURN $COLLECTION))
e1 e2 ... en
(IF terminal condition THEN terminal action
   (RETURN $COLLECTION))
(IF collection condition THEN (SETQ ←COLLECTION
                               (CONS item $COLLECTION)))
iteration step
(FAIL)   .
```

The nondeterministic expression may be any QA4 expression. Normally, the REPEAT statement is intended to gather information by the use of side effects. Therefore, the FAIL at the conclusion of each step is only partly backtracking. If the FAIL backtracks to any place in the body, conditions, or step it is treated as a normal failure. However, if any FAIL backtracks to the nondeterministic expression, full

backtracking does not occur. Side effects are not removed, but the next alternative is chosen and variables are rebound under the old backtracking context, and the body is again executed.

3. Options

- terminal-condition

  The terminal-condition can take the form

  UNTIL p

  or the form

  WHILE p

  where p is a predicate expression. If a terminal-condition is present it may be followed by a terminal action that takes the form

  THEN e .

  If the condition causes the termination of the REPEAT, the expression of the action is performed just before the final exit from the statement.

- Collection

  The collection condition and the item provide a method of gathering a single data object during each step of the iteration. This clause takes the form

  COLLECT e WHEN p .

If it is present, p is tested just after each execution of the REPEAT body. If it holds, e is appended to a special tuple kept by the interpreter. If p is not present, e is always appended to the tuple. This tuple becomes the value returned at the completion of the REPEAT.

- <u>iteration-step</u>

  The <u>iteration-step</u> takes the form

  STEP e .

  It is executed just before the condition FAIL is given.

- <u>final failure</u>

  The final failure expression is executed if the cases of the nondeterministic expression have been exhausted and the REPEAT termination condition still does not hold. It takes the form

  FINALLY e .

B. <u>FIND Statement</u>

The FIND statement is a macro expansion of the REPEAT statement. The form of the statement is

(FIND BETWEEN <u>lower</u> <u>upper</u> <u>item</u>

DO e1 e2 ... en) .

Its definition in terms of the REPEAT is

```
(REPEAT (WHILE (LTQ $NUMBER upper))
    (COLLECT item)
    (STEP (SETQ ←NUMBER (PLUS $NUMBER 1)))
    (FINALLY (IF (LT lower $NUMBER) THEN (FAIL)))
    DO e1 e2 ... en)   .
```

For example, the following statement finds three boxes that contain green blocks:

```
(FIND BETWEEN 1 3 X DO
    (GOAL $PROVE  (BOX ←X))
    (GOAL $PROVE  (CONTAINS $X ←B))
    (GOAL $PROVE  (GREEN $B))) .
```

Chapter Five

EXPECTATIONS AND REFLECTIONS

I PROJECTS

A. Imhotep

There will be, during the next year, a number of problem-solving systems programmed in QA4. Each of these will deal with its own, highly specialized problems. But one robot project will attempt to be a focal point of QA4 development, incorporating innovations as they arise and even using large portions of the other projects as subsystems.

Our robot, "Imhotep," will be a simulated arm that can stack blocks. We have chosen the problem of building a house as the long-range goal of our effort. The following scenario demonstrates the research problems we wish to explore.

Imhotep will begin with detailed initial knowledge in the following subject areas:

- Primitive arm movements, e.g., move to location (X Y Z).
- Intermediate-level actions, e.g., pick up a block.
- Facts about the world, e.g., block1 is on top of stack3.
- Common sense reasoning, a simple theorem prover.
- Program synthesis, a program that writes primitive QA4 programs.
- The world, e.g., programs that simulate the arm and that simulate falling blocks.

All this knowledge will have to exist as QA4 programs. That is, each of these packages will be a QA4 problem solver that can work on certain kinds of goals. (The design, implementation, and refinements of these packages will require most of the effort on this task during the coming year.) Not included in this initial knowledge will be concepts of stacks, building, walls, lintels, roofs, corners, and other notions that will arise during the scenario.

Step 1: We ask Imhotep to build a stack. He responds that he does not understand "build" or "stack." We supply more precise definitions, and eventually a stack program is synthesized. The program is executed, and the results are evaluated. If we are not satisfied, we express our complaints, and Imhotep modifies the program accordingly. A most important point during this process is that all actions are goal-directed, and all inputs to the system are in response to a question by the system. Imhotep accepts input only to resolve an internal question or test a hypothesis.

Step 2: Build a lintel. This is also a program-synthesis problem, but now the stack-building program may be used as a subroutine. This will be evident from the definition of a lintel: two stacks with a span ... .

Step 3: Build a wall. Here we expect a wall to have interlocking blocks and flush ends. We also expect the wall-building program to be synthesized by using information developed during the solution of Step 1. This might be more appropriately called program modification, for it involves changing an existing program to meet new requirements. Blocks of various sizes will be available.

Step 4: Build a wall with a lintel. For this problem we wish to coordinate the wall and lintel programs, using them together without writing new programs. This will be by far the most difficult task, for it will require coordinating two independent processes so as to produce a more elaborate result than either would product alone.

Step 5: Build two more walls.

Step 6: Add a roof. A roof is a stack of long blocks resting on its side on top of the walls.

When the program has achieved these steps, it will have sucessfully built a house. As can be seen, this project will be primarily directed toward program synthesis and the use of QA4 as an aid in the development of problem-solving programs. This effort will also emphasize the useful integration of new programs into the systen, and their subsequent modification.

B. Other Projects

The other projects, most of them underway, include:

- A program verifier that will be used to automatically verify the correctness of small programs. It will also operate in an interactive mode, permitting the user to guide the process of verifying large programs (Elspas, 1972). The verifier will stress expression simplification as an inference rule.
- A program synthesizer, similar to the verifier, that will automatically synthesize small programs and interactively synthesize large programs (Manna and Waldinger, 1971). The synthesizer will stress the coordinated use of quantified logical statements, procedural definitions, and pragmatic advice for its program construction. It will also stress the development of the internal structure of programs, unlike Imhotep which is primarily concerned with the usefulness of the finished programs.
- A robot planner that will generate plans similar to those required of Imhotep, but will actually perform the plans, recover from errors of either reasoning or execution, gain knowledge about the world during plan execution, and replan intelligently. This system will stress pseudo-parallel searching, plan construction, and the recognition of similar problems.

- A vision system that will direct its analysis of scenes in a goal-oriented fashion with the intent of deducing relational information about its environment. This system will stress the use of large amounts of relational information, especially using sets and bags, to deduce relevant features and operations that, in turn, will guide the top-down analysis programs.

As we have pointed out, each of the projects emphasizes a different problem area and each will stress different facets of QA4. Since they all strongly share a common base, however, they blend well and each will continually contribute to the success of the others.

II EXTENSIONS

A. Lamarckian Evolution

Much of the character of the QA4 language has resulted from gradual changes, introduced to aid or enhance systems as they were being programmed. Automatic backtracking exemplifies how the summation of minute changes results in a qualitative difference in the character of the language. We first believed that automatic backtracking would be especially useful for searching small finite sets. The effect, however, produced comic debugging sessions where misspelled variable names or slightly incorrect program forms initiated automatic backtracking, which, in turn, led to the belief that the models were incorrect or that the goal direction had gone awry. Eventually, backtracking was made optional and the standard default for the interpreter was to respond to failures as errors. Slowly, one program form at a time, all the automatic backtracking disappeared, so that now the programmer must explicitly state where backtracking is to occur. Thus, while the fundamental power of the language remained constant, the style of programming took a complete turnaround.

A similar experience has occurred with the GOAL statements. Originally all programs could be applied to a goal and only their bound variable patterns would disqualify them. Then the system was changed

so that programs could be classified into goal classes. There were special statements in the language to classify the programs, and an especially obscure method was implemented to test if goals were "satisfied" before programs were chosen from the goal classes. These statements were finally discarded. An optional form of restrictions was added to the EXISTS and GOAL statements, and the goal class method was changed to the current tuple format.

When the current restriction format was substituted for the obscure "satisfied" programs, the idea of a relevant context developed. This led to the current WRT semantics. Since the tuple of relevant programs has replaced the goal class, we have begun to see ways to save the tuples used in a particular problem solution. The saved tuples could be reordered so that the successful programs--i.e., the ones that satisfied the goals--are first. This set of reordered tuples can be saved as a "plan." Later, when a similar problem arises, the "plan" will direct the problem solver directly to a solution except, of course, at the points where the new problem differs from the original.

Similar evolution has occurred with other facets of the system. Currently, a convention of quantified expressions and goals is being developed. We expect that, in the long run, each of these small changes will result in a qualitatively different and more intuitive language.

B. Searching for Goal Solutions

The most dramatic change we anticipate will be in the semantics of the goal statements. During some kinds of robot planning, the depth-first orientation of the goal mechanism appears to be a major deterrent to speedy, succesful planning. There are many cases, route finding for example, when pseudo-parallel searches appear to economize effort rather than waste it. If we think of the search as a growing graph, then our pseudo-parallel search means that we are saving nodes and later returning to them for further expansion. Within the QA4 framework, however, the nodes arise from goal statements and backtracking is not a satisfactory method for conducting the search.

Instead, each goal statement should spawn processes for the programs that can satisfy the goal. Each process should run until it reaches a point where its utility can be approximated. At that point, a RESUME statement in the process can return control to the mother process. The mother process may then alternate among the programs working on the goal, always choosing the best. If the mother was originally given a bound, and each child exceeds the bound, the mother may, in turn, RESUME its mother with an estimate of success. While each child is running, moreover, it may execute goal statements that either generate more processes or search in the standard goal fashion.

With this framework we have preserved the basic QA4 approach of using procedures as models and have introduced the facility for each goal statement to search according to the method appropriate for it. Estimates of success are determined on a local basis and are computed by the procedure doing the search, eliminating the almost impossible task of writing programs that examine another program state to determine its progress. We must yet make the searching operate smoothly and default in natural ways. We must also work out techniques for approximating progress. But we have designed a robot planner based on this type of goal statement, and it appears to be a considerable advancement over current planners.

C. Transitive Relations

The idea behind the automatic handling of equivalence relations can be extended to transitive relations as well. Within many problem-solving tasks, goals are mainly concerned with small sets of objects and their relations (Elliott, 1965). For example, robot planners may be concerned with a few rooms and a few blocks together with relations such as next-to, left-of, and between.

There are many properties that transitive relations satisfy. For instance, they may be reflexive or irreflexive; symmetric, antisymmetric, or asymmetric; functional in the first argument or functional in the second argument. Combinations of relations may form

groups. For example <, ≤, >, and ≥, form a group of order four under the operations of negation and argument reversal. Composition of relations also satisfies algebraic properties such as absorption ($X < Y \Rightarrow X + Z < Y + Z$) and adsorption.

If the user could identify a collection of relations, state both their individual properties and the relations between them, a system could be written that would mirror the equivalence relation mechanism. Statements of the form

(FA X IN A) (EX Y IN B) (R X Y)

could be assimilated into a storage network. Space could be kept at a minimum and questions about the specific relations of objects could easily be answered. Statements using different but related relations would be accommodated into a single unified structure.

A system similar to the equivalence relations statements could be made to operate within the QA4 language. Such a system could greatly enhance the problem-solving ability of systems that deal with many sometimes complex relations between small sets of objects.

D. Efficiencies

The QA4 user community is expanding beyond the small group that has built the system. And with this growth comes the expected complaints. The interpreter is too large, as are all systems that govern. The services consume too much of the available resources. The net, for example, takes up too much storage and the system is unresponsive.

The pattern matcher is too slow, and the system programmers do not react quickly enough to user demands. The solutions, of course, are in the promises for the future.

There are great reductions that can be achieved with the net. As discussed in Appendix II, a decrease of about a factor of 10 in storage is easily achievable. The net programs currently use the extra space to keep statistics on each path followed by every reference to the net. When we are convinced that the distribution is substantially uniform for large problems, the monitoring can be removed and the savings realized.

The pattern matcher suffers the effects of a more fundamental deficiency. With many QA4 patterns there are alternative matches. Sets, bags, and fragment variables are the sources of the alternatives. When the pattern matcher is invoked and a match is found, information must be saved so that, later, the matcher can be restarted to find the next alternative match. Not only is this saving of information costly, but the matcher itself is written in a clumsy manner with many extra programs to perform the restarting. If LISP had a process structure (Bobrow and Wegbreit, 1972), so that the matcher could be run as a coroutine and the restart information could easily be saved as, say, a stack segment, significant savings could be realized in both the size of the pattern matcher and the amount of restart information. Moreover, the internal structure would be more clear and straightforward.

Finally, if LISP permitted the process structure programming that would benefit the pattern matcher, the QA4 interpreter could be abolished. When backtracking, the state of variables and data structure within QA4 is restored with the context mechanism. The only information not restorable is the state of the QA4 interpreter--that is, LISP's push down stack. If this information were savable, then QA4 programs could be LISP programs that called functions such as GOAL and ASSERT. This would result in many improvements.

QA4 programs would not be in the net, reducing the size of the net, saving large amounts of storage for QA4 systems, and reducing the net search time.

The QA4 interpreter would disappear, and with it would go about 25 percent of the code of the QA4 system. More importantly, the list structures built by the QA4 interpreter to evaluate QA4 programs, the ones that serve the function of savable stacks, would then be LISP's savable stacks. Those CONSES that build the stacks account for about 30 percent of the total CONSES during QA4 evaluation.

And finally, if QA4 programs were LISP programs they could be compiled. There is no doubt that this could result in at least a speed-up factor of 100 and maybe even more. There are a few problems in making QA4 programs LISP programs, but they are all minor. All in all, it efficiently becomes a limiting factor for QA4 users; the solution, as opposed to stopgap measures, is to introduce process structures into LISP.

III TRENDS IN QA4 PROBLEM SOLVERS

A. Backtracking

The shift away from backtracking is the most surprising trend in QA4 programming. What we originally thought would be a heavily used feature has come to play an important but different role. In almost all cases where backtracking offers a solution to a programming problem, other solutions are available. Moreover, it usually introduces more problems than it is worth. When one attempts to use it for small local loops, it tends to leave unexpected backtracking points in the execution path, and these inevitably interfere with the global strategies. To program them away with the ATTEMPT statement or other similar programming techniques often introduces yet another layer of interference in the global strategies. Even the potential problem of misspelled identifiers in patterns causes havoc during debugging.

Thus, the trend is away from backtracking for small local problems. We are, however, discovering its use for problems more deeply rooted in the models and problems. That is, backtracking seems particularly suited as a means by which a problem solver can simplify a major problem. For example, it may have to solve a complicated problem and has three blocks that may work. Solving the problem by supposing one specific block will do and then attempting to find the solution with that concrete example is often a good strategy, and backtracking on

that level provides a way of easily implementing the strategy. The skills of programming with appropriate backtracking will, we hope, develop as our problem-solvers evolve.

Even with highly restricted use of backtracking, however, there is a nagging worry that computational power is being wasted. For in standard backtracking, all side effects of a computation are removed. If the search with the second alternative is to use information gleaned from the search with the first alternative, special care must be taken to ensure that the information is available after the backtracking has occurred. This requires considerable additional effort in the design of programs to work on goals--so much, in fact, that when one attempts it, one finally writes programs that consider all the alternatives simultaneously. Thus backtracking still appears as a large unanswered question. Just when it is appropriate seems to be a matter of taste and a question of tradeoffs--clear concise or easily written programs versus more efficient but sometimes complicated programs.

B. Efficiency

There has finally begun to develop, among QA4 programmers, less of a concern for minor efficiencies and more concern about the nature of the searching. If the problem-solving system is large, complicated, and still developing and a speed-up factor of 10 is needed, then the system needs better strategies. Faster matches, interpreter, storage

- The message is received by A, and 2 becomes the value A's RESUME statement, and A continues execution. Thus each RESUME statement serves two purposes: it sends a message and receives a message in return.

T4
- A executes a RESUME statement and sends a 3 to B.
- A becomes suspended and B begins execution with 3 as the value of its RESUME statement.

2. Form

The form of the RESUME statement is

(RESUME process message)

where process instantiates to a QA4 process and message is the expression that is instantiated and sent to the process. When a RESUME statement is executed, its process is suspended. The suspension is made in such a way that a value is required as the value of the RESUME statement before the process can be restarted. This value can be supplied by a RESUME statement from any other process in the system. Normally, process is a variable that is the value of an INCARNATE statement. Thus, processes are linked together merely because they have variables that point to other processes. This means that there is no rigid structure to QA4 processes as there often is in systems that require pipes or plugs and sockets. Moreover, the interconnections can be dynamic and continually change within a QA4 process structure.

B. INCARNATE Statement

Suppose that F and G are functions. The following lines of program may have created the above example.

(SETQ ←A (INCARNATE $F))

(SETQ ←B (INCARNATE $G))

(RESUME $A 0) .

The form of the INCARNATE statement is

(INCARNATE *e*)

where *e* instantiates to a lambda expression. The value of the INCARNATE statement is a process that needs a value to begin execution. The lambda expression indicated by *e* will be applied to the message from the first RESUME statement that resumes the process. For example, suppose F has the definition

(LAMBDA (Q ←X) ..)

and we execute

(SETQ ←P (INCARNATE $F))

(RESUME $P (Q 7)) .

e of the INCARNATE statement is $F and this instantiates to the LAMBDA definition of F. The value of the INCARNATE statement is a process, and P is bound to that process. The RESUME statement supplies the *message* (Q 7) to the process. Since this is the first resume to the process, the lambda expression is applied to the *message*. That is, (Q ←X) is bound to (Q 7) and the process is started.

A particular LAMBDA expression can be incarnated many times. We could, for example, have a program that reads:

```
(SETQ ←C (INCARNATE $F))
(SETQ ←D (INCARNATE $F))    .
```

C and D are different processes with different contexts. They share the same code but will have different variable bindings. Since this multiple incarnation is possible, references to processes must be made via the process and not the function name.

C. CONNECT Statement

The CONNECT statement has the form

(CONNECT _p_ _a_ _q_ _b_)

where _a_ and _b_ are variables that are bound to processes and _p_ and _q_ are identifiers. The statement assigns _p_ the value of _b_ under _a_'s context, and it assigns _q_ the value of _a_ under _b_'s context. For example, suppose we execute

```
(SETQ ←A (INCARNATE $F))
(SETQ ←B (INCARNATE $G))
(CONNECT P $A Q $B)    .
```

The following diagram shows the situation after the CONNECT statement. Arrows indicate values, circles indicate processes, variables within circles indicate what the value is with respect to the context of the process.

D. WAIT Statement

A newly incarnated process is called the child of the process that incarnated it, and we refer to the creating process as the mother of its child. Many times a child wants to RESUME to its mother, but lacks a way of referring to her. The WAIT statement provides a mechanism for this.

(WAIT message)

operates just like a RESUME to the mother process, but does not require that the mother process be explicitly mentioned.

Once a process has been created it may be passed between functions just like any other data object. It can thus take on a life independent of its mother. This violates ALGOL scope conventions and normal stack operations. When all references to a process are removed, the process automatically disappears. That is, when there are no longer any variables that are bound to a process, the garbage collector will come and get it.

XII ITERATION STATEMENTS

A. REPEAT Statement

1. Example

In many languages, iteration statements are intended to step through indexed data structures: FOR in ALGOL and DO in FORTRAN for example. In QA4, however, they are intended to enhance the utility of the nondeterminism of the language. As a result, they operate through the failure mechanism rather than simple transfers. They provide two features: iteration through data structures or multiple pattern matches without the loss of side effects, and collection of items at each iteration step. For example,

```
(REPEAT (EXISTS (PAND ←Y (ON-TOP-OF ←X BRICK1)))
     DO ($PICKUP $X)
          ($PLACE-IN $X BLUE-BOX))
```

is a statement that puts all the bricks on BRICK1 in the blue box. REPEAT is the major statement, and all the other iteration statements are defined in terms of it. Some of the options of the REPEAT are included to facilitate the definition of the other statements and may not appear useful.

2. Form

The form of the REPEAT statement is

(REPEAT <u>nde</u> <u>options</u> DO <u>e1</u> <u>e2</u> ... <u>en</u>)

where *nde* is a nondeterministic expression, each *e* is an expression, and *options* are pairs of keywords and expressions. The options are:

- A terminal condition and a terminal action
- A collection condition and an item
- An iteration step
- A final failure expression.

These will be discussed more fully below. The execution of the REPEAT may be described by the following pseudo-QA4 program:

```
(SETQ ←COLLECTION (TUPLE))
(FAILING? failure expression (RETURN $COLLECTION))
e1 e2 ... en
(IF terminal condition THEN terminal action
   (RETURN $COLLECTION))
(IF collection condition THEN (SETQ ←COLLECTION
                                (CONS item $COLLECTION)))
iteration step
(FAIL)   .
```

The nondeterministic expression may be any QA4 expression. Normally, the REPEAT statement is intended to gather information by the use of side effects. Therefore, the FAIL at the conclusion of each step is only partly backtracking. If the FAIL backtracks to any place in the body, conditions, or step it is treated as a normal failure. However, if any FAIL backtracks to the nondeterministic expression, full

backtracking does not occur. Side effects are not removed, but the next alternative is chosen and variables are rebound under the old backtracking context, and the body is again executed.

3. Options

- terminal-condition

  The terminal-condition can take the form

  UNTIL p

  or the form

  WHILE p

  where p is a predicate expression. If a terminal-condition is present it may be followed by a terminal action that takes the form

  THEN e .

  If the condition causes the termination of the REPEAT, the expression of the action is performed just before the final exit from the statement.

- Collection

  The collection condition and the item provide a method of gathering a single data object during each step of the iteration. This clause takes the form

  COLLECT e WHEN p .

If it is present, p is tested just after each execution of the REPEAT body. If it holds, e is appended to a special tuple kept by the interpreter. If p is not present, e is always appended to the tuple. This tuple becomes the value returned at the completion of the REPEAT.

- iteration-step

  The iteration-step takes the form

  STEP e .

  It is executed just before the condition FAIL is given.

- final failure

  The final failure expression is executed if the cases of the nondeterministic expression have been exhausted and the REPEAT termination condition still does not hold. It takes the form

  FINALLY e .

B. FIND Statement

The FIND statement is a macro expansion of the REPEAT statement. The form of the statement is

```
(FIND BETWEEN lower upper item
      DO e1 e2 ... en)   .
```

Its definition in terms of the REPEAT is

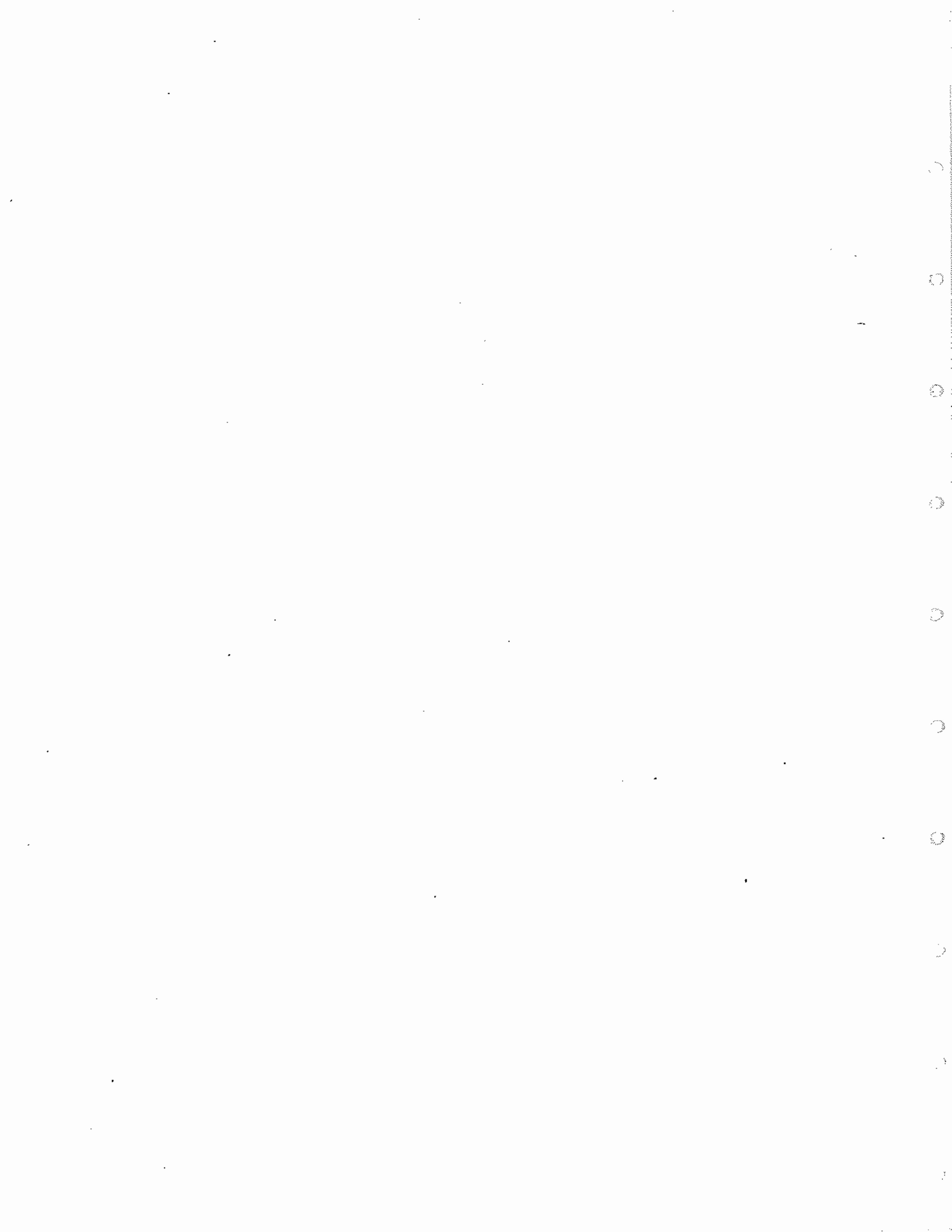
```
(REPEAT (WHILE (LTQ $NUMBER upper))
     (COLLECT item)
     (STEP (SETQ ←NUMBER (PLUS $NUMBER 1)))
     (FINALLY (IF (LT lower $NUMBER) THEN (FAIL)))
     DO e1 e2 ... en)    .
```

For example, the following statement finds three boxes that contain green blocks:

```
(FIND BETWEEN 1 3 X DO
     (GOAL $PROVE  (BOX ←X))
     (GOAL $PROVE  (CONTAINS $X ←B))
     (GOAL $PROVE  (GREEN $B))) .
```

Chapter Five

EXPECTATIONS AND REFLECTIONS



CHAPTER FIVE--EXPECTATIONS AND REFLECTIONS

I PROJECTS

A. Imhotep

There will be, during the next year, a number of problem-solving systems programmed in QA4. Each of these will deal with its own, highly specialized problems. But one robot project will attempt to be a focal point of QA4 development, incorporating innovations as they arise and even using large portions of the other projects as subsystems.

Our robot, "Imhotep," will be a simulated arm that can stack blocks. We have chosen the problem of building a house as the long-range goal of our effort. The following scenario demonstrates the research problems we wish to explore.

Imhotep will begin with detailed initial knowledge in the following subject areas:

- Primitive arm movements, e.g., move to location (X Y Z).
- Intermediate-level actions, e.g., pick up a block.
- Facts about the world, e.g., block1 is on top of stack3.
- Common sense reasoning, a simple theorem prover.
- Program synthesis, a program that writes primitive QA4 programs.
- The world, e.g., programs that simulate the arm and that simulate falling blocks.

All this knowledge will have to exist as QA4 programs. That is, each of these packages will be a QA4 problem solver that can work on certain kinds of goals. (The design, implementation, and refinements of these packages will require most of the effort on this task during the coming year.) Not included in this initial knowledge will be concepts of stacks, building, walls, lintels, roofs, corners, and other notions that will arise during the scenario.

Step 1: We ask Imhotep to build a stack. He responds that he does not understand "build" or "stack." We supply more precise definitions, and eventually a stack program is synthesized. The program is executed, and the results are evaluated. If we are not satisfied, we express our complaints, and Imhotep modifies the program accordingly. A most important point during this process is that all actions are goal-directed, and all inputs to the system are in response to a question by the system. Imhotep accepts input only to resolve an internal question or test a hypothesis.

Step 2: Build a lintel. This is also a program-synthesis problem, but now the stack-building program may be used as a subroutine. This will be evident from the definition of a lintel: two stacks with a span ... .

Step 3: Build a wall. Here we expect a wall to have interlocking blocks and flush ends. We also expect the wall-building program to be synthesized by using information developed during the solution of Step 1. This might be more appropriately called program modification, for it involves changing an existing program to meet new requirements. Blocks of various sizes will be available.

Step 4: Build a wall with a lintel. For this problem we wish to coordinate the wall and lintel programs, using them together without writing new programs. This will be by far the most difficult task, for it will require coordinating two independent processes so as to produce a more elaborate result than either would product alone.

Step 5: Build two more walls.

Step 6: Add a roof. A roof is a stack of long blocks resting on its side on top of the walls.

When the program has achieved these steps, it will have sucessfully built a house. As can be seen, this project will be primarily directed toward program synthesis and the use of QA4 as an aid in the development of problem-solving programs. This effort will also emphasize the useful integration of new programs into the systen, and their subsequent modification.

B. <u>Other Projects</u>

The other projects, most of them underway, include:

- A program verifier that will be used to automatically verify the correctness of small programs. It will also operate in an interactive mode, permitting the user to guide the process of verifying large programs (Elspas, 1972). The verifier will stress expression simplification as an inference rule.
- A program synthesizer, similar to the verifier, that will automatically synthesize small programs and interactively synthesize large programs (Manna and Waldinger, 1971). The synthesizer will stress the coordinated use of quantified logical statements, procedural definitions, and pragmatic advice for its program construction. It will also stress the development of the internal structure of programs, unlike Imhotep which is primarily concerned with the usefulness of the finished programs.
- A robot planner that will generate plans similar to those required of Imhotep, but will actually perform the plans, recover from errors of either reasoning or execution, gain knowledge about the world during plan execution, and replan intelligently. This system will stress pseudo-parallel searching, plan construction, and the recognition of similar problems.

- A vision system that will direct its analysis of scenes in a goal-oriented fashion with the intent of deducing relational information about its environment. This system will stress the use of large amounts of relational information, especially using sets and bags, to deduce relevant features and operations that, in turn, will guide the top-down analysis programs.

As we have pointed out, each of the projects emphasizes a different problem area and each will stress different facets of QA4. Since they all strongly share a common base, however, they blend well and each will continually contribute to the success of the others.

II EXTENSIONS

A. Lamarckian Evolution

Much of the character of the QA4 language has resulted from gradual changes, introduced to aid or enhance systems as they were being programmed. Automatic backtracking exemplifies how the summation of minute changes results in a qualitative difference in the character of the language. We first believed that automatic backtracking would be especially useful for searching small finite sets. The effect, however, produced comic debugging sessions where misspelled variable names or slightly incorrect program forms initiated automatic backtracking, which, in turn, led to the belief that the models were incorrect or that the goal direction had gone awry. Eventually, backtracking was made optional and the standard default for the interpreter was to respond to failures as errors. Slowly, one program form at a time, all the automatic backtracking disappeared, so that now the programmer must explicitly state where backtracking is to occur. Thus, while the fundamental power of the language remained constant, the style of programming took a complete turnaround.

A similar experience has occurred with the GOAL statements. Originally all programs could be applied to a goal and only their bound variable patterns would disqualify them. Then the system was changed

so that programs could be classified into goal classes. There were special statements in the language to classify the programs, and an especially obscure method was implemented to test if goals were "satisfied" before programs were chosen from the goal classes. These statements were finally discarded. An optional form of restrictions was added to the EXISTS and GOAL statements, and the goal class method was changed to the current tuple format.

When the current restriction format was substituted for the obscure "satisfied" programs, the idea of a relevant context developed. This led to the current WRT semantics. Since the tuple of relevant programs has replaced the goal class, we have begun to see ways to save the tuples used in a particular problem solution. The saved tuples could be reordered so that the successful programs--i.e., the ones that satisfied the goals--are first. This set of reordered tuples can be saved as a "plan." Later, when a similar problem arises, the "plan" will direct the problem solver directly to a solution except, of course, at the points where the new problem differs from the original.

Similar evolution has occurred with other facets of the system. Currently, a convention of quantified expressions and goals is being developed. We expect that, in the long run, each of these small changes will result in a qualitatively different and more intuitive language.

B. Searching for Goal Solutions

The most dramatic change we anticipate will be in the semantics of the goal statements. During some kinds of robot planning, the depth-first orientation of the goal mechanism appears to be a major deterrent to speedy, succesful planning. There are many cases, route finding for example, when pseudo-parallel searches appear to economize effort rather than waste it. If we think of the search as a growing graph, then our pseudo-parallel search means that we are saving nodes and later returning to them for further expansion. Within the QA4 framework, however, the nodes arise from goal statements and backtracking is not a satisfactory method for conducting the search.

Instead, each goal statement should spawn processes for the programs that can satisfy the goal. Each process should run until it reaches a point where its utility can be approximated. At that point, a RESUME statement in the process can return control to the mother process. The mother process may then alternate among the programs working on the goal, always choosing the best. If the mother was originally given a bound, and each child exceeds the bound, the mother may, in turn, RESUME its mother with an estimate of success. While each child is running, moreover, it may execute goal statements that either generate more processes or search in the standard goal fashion.

With this framework we have preserved the basic QA4 approach of using procedures as models and have introduced the facility for each goal statement to search according to the method appropriate for it. Estimates of success are determined on a local basis and are computed by the procedure doing the search, eliminating the almost impossible task of writing programs that examine another program state to determine its progress. We must yet make the searching operate smoothly and default in natural ways. We must also work out techniques for approximating progress. But we have designed a robot planner based on this type of goal statement, and it appears to be a considerable advancement over current planners.

C. Transitive Relations

The idea behind the automatic handling of equivalence relations can be extended to transitive relations as well. Within many problem-solving tasks, goals are mainly concerned with small sets of objects and their relations (Elliott, 1965). For example, robot planners may be concerned with a few rooms and a few blocks together with relations such as next-to, left-of, and between.

There are many properties that transitive relations satisfy. For instance, they may be reflexive or irreflexive; symmetric, antisymmetric, or asymmetric; functional in the first argument or functional in the second argument. Combinations of relations may form

groups. For example $<$, $\leq$, $>$, and $\geq$, form a group of order four under the operations of negation and argument reversal. Composition of relations also satisfies algebraic properties such as absorption ($X < Y \Rightarrow X + Z < Y + Z$) and adsorption.

If the user could identify a collection of relations, state both their individual properties and the relations between them, a system could be written that would mirror the equivalence relation mechanism. Statements of the form

(FA X IN A) (EX Y IN B) (R X Y)

could be assimilated into a storage network. Space could be kept at a minimum and questions about the specific relations of objects could easily be answered. Statements using different but related relations would be accommodated into a single unified structure.

A system similar to the equivalence relations statements could be made to operate within the QA4 language. Such a system could greatly enhance the problem-solving ability of systems that deal with many sometimes complex relations between small sets of objects.

D. Efficiencies

The QA4 user community is expanding beyond the small group that has built the system. And with this growth comes the expected complaints. The interpreter is too large, as are all systems that govern. The services consume too much of the available resources. The net, for example, takes up too much storage and the system is unresponsive.

The pattern matcher is too slow, and the system programmers do not react quickly enough to user demands. The solutions, of course, are in the promises for the future.

There are great reductions that can be achieved with the net. As discussed in Appendix II, a decrease of about a factor of 10 in storage is easily achievable. The net programs currently use the extra space to keep statistics on each path followed by every reference to the net. When we are convinced that the distribution is substantially uniform for large problems, the monitoring can be removed and the savings realized.

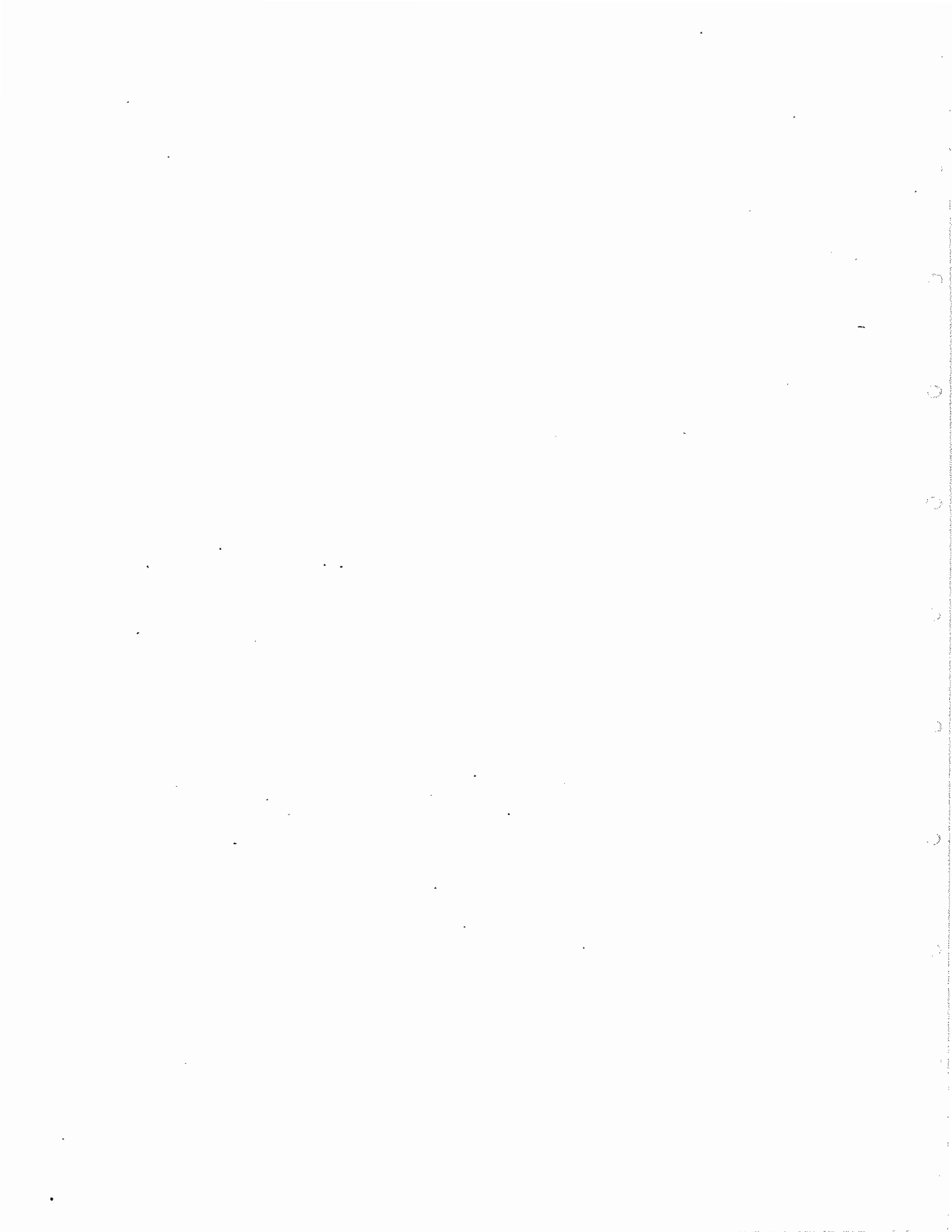
The pattern matcher suffers the effects of a more fundamental deficiency. With many QA4 patterns there are alternative matches. Sets, bags, and fragment variables are the sources of the alternatives. When the pattern matcher is invoked and a match is found, information must be saved so that, later, the matcher can be restarted to find the next alternative match. Not only is this saving of information costly, but the matcher itself is written in a clumsy manner with many extra programs to perform the restarting. If LISP had a process structure (Bobrow and Wegbreit, 1972), so that the matcher could be run as a coroutine and the restart information could easily be saved as, say, a stack segment, significant savings could be realized in both the size of the pattern matcher and the amount of restart information. Moreover, the internal structure would be more clear and straightforward.

Finally, if LISP permitted the process structure programming that would benefit the pattern matcher, the QA4 interpreter could be abolished. When backtracking, the state of variables and data structure within QA4 is restored with the context mechanism. The only information not restorable is the state of the QA4 interpreter--that is, LISP's push down stack. If this information were savable, then QA4 programs could be LISP programs that called functions such as GOAL and ASSERT. This would result in many improvements.

QA4 programs would not be in the net, reducing the size of the net, saving large amounts of storage for QA4 systems, and reducing the net search time.

The QA4 interpreter would disappear, and with it would go about 25 percent of the code of the QA4 system. More importantly, the list structures built by the QA4 interpreter to evaluate QA4 programs, the ones that serve the function of savable stacks, would then be LISP's savable stacks. Those CONSES that build the stacks account for about 30 percent of the total CONSES during QA4 evaluation.

And finally, if QA4 programs were LISP programs they could be compiled. There is no doubt that this could result in at least a speed-up factor of 100 and maybe even more. There are a few problems in making QA4 programs LISP programs, but they are all minor. All in all, it efficiently becomes a limiting factor for QA4 users; the solution, as opposed to stopgap measures, is to introduce process structures into LISP.



III TRENDS IN QA4 PROBLEM SOLVERS

A. Backtracking

The shift away from backtracking is the most surprising trend in QA4 programming. What we originally thought would be a heavily used feature has come to play an important but different role. In almost all cases where backtracking offers a solution to a programming problem, other solutions are available. Moreover, it usually introduces more problems than it is worth. When one attempts to use it for small local loops, it tends to leave unexpected backtracking points in the execution path, and these inevitably interfere with the global strategies. To program them away with the ATTEMPT statement or other similar programming techniques often introduces yet another layer of interference in the global strategies. Even the potential problem of misspelled identifiers in patterns causes havoc during debugging.

Thus, the trend is away from backtracking for small local problems. We are, however, discovering its use for problems more deeply rooted in the models and problems. That is, backtracking seems particularly suited as a means by which a problem solver can simplify a major problem. For example, it may have to solve a complicated problem and has three blocks that may work. Solving the problem by supposing one specific block will do and then attempting to find the solution with that concrete example is often a good strategy, and backtracking on

that level provides a way of easily implementing the strategy. The skills of programming with appropriate backtracking will, we hope, develop as our problem-solvers evolve.

Even with highly restricted use of backtracking, however, there is a nagging worry that computational power is being wasted. For in standard backtracking, all side effects of a computation are removed. If the search with the second alternative is to use information gleaned from the search with the first alternative, special care must be taken to ensure that the information is available after the backtracking has occurred. This requires considerable additional effort in the design of programs to work on goals--so much, in fact, that when one attempts it, one finally writes programs that consider all the alternatives simultaneously. Thus backtracking still appears as a large unanswered question. Just when it is appropriate seems to be a matter of taste and a question of tradeoffs--clear concise or easily written programs versus more efficient but sometimes complicated programs.

B. Efficiency

There has finally begun to develop, among QA4 programmers, less of a concern for minor efficiencies and more concern about the nature of the searching. If the problem-solving system is large, complicated, and still developing and a speed-up factor of 10 is needed, then the system needs better strategies. Faster matches, interpreter, storage

retrieval programs, or what have you, are not the solution. When a search problem can be concisely stated and algorithm search methods are available, then tight code will always help. But the problems to be solved by QA4 problem-solvers do not have that flavor. This attitude is held by everyone working with QA4 and is reflected in the effort they put into the overall structure of their problem-solvers.

C. Procedural and Declarative Language Development

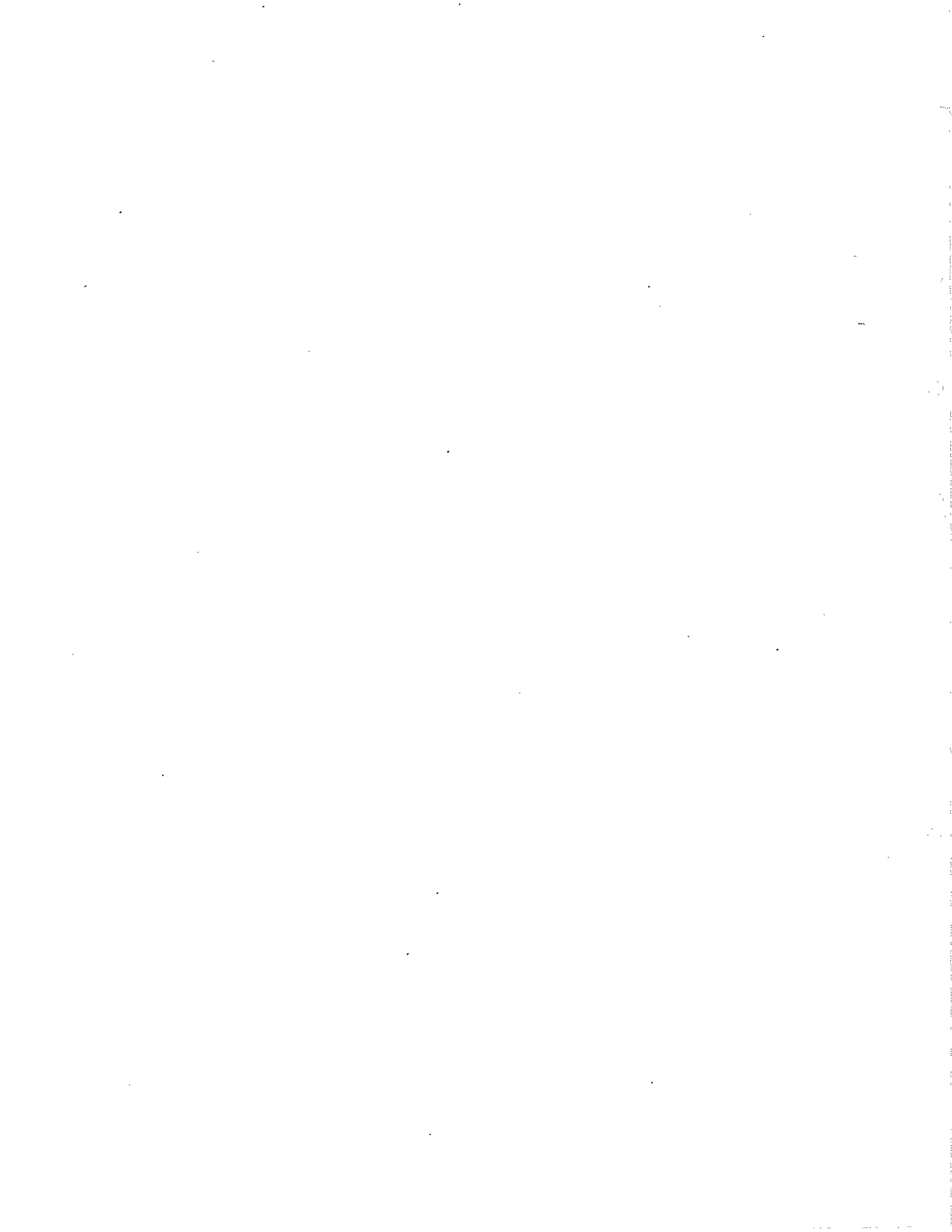
The mixing of procedural and declarative knowledge within QA4 is constantly becoming richer. With the realization that many apparently declarative statements can also be considered procedural, QA4 users are beginning to blur the distinction. Thus we see QA4 programs that have both quantified assertions and programs that may be used by GOAL statements. For example, one might develop during a problem solution, the facts (P ←X A) and (P ←X B). The system may also have programs that work on goal expressions of the form (P ←X ←Y). In this way, a goal statement may sometimes be satisfied by the quantified expressions and at other times by the program. So, instead of stressing either the logical or the procedural formulation of knowledge, QA4 programs are tending to thoroughly mix the two. This mixture, moreover, is arising because we are making both available.

D. Ad Hoc or General

The development of QA4 and the problem-solvers being constructed with it is a clear shift away from the idea of a general problem-solver or question-answering system. How far the shift can go before the systems are though of as ad hoc is a question of taste. What is significant about this shift, however, is that much of the emphasis of the research has shifted to the problem domains. Instead of attempting to fit a problem into a syntactic framework that could be digested and manipulated by some arbitrary reasoning method or programming language such as resolution logic or GPS, the researchers working with QA4 are free to manipulate the problem and the solution simultaneously. This new latitude has led to research into the nature of program synthesis and the relational structure of robot models. This style of problem formulation will, in the end, overcome the suggestions of being ad hoc and lead more directly to the invention of intelligent systems than the search for a single universal solution.

Appendix I

LISTING OF THE ROBOT SYSTEM



THE QA4 OPERATORS: THE GOTHRUDOOR OPERATOR

```
[RPAQQ GOTHRUDOOR (LAMBDA (INROOM (TUPLE ROBOT ←M))
        (PROG (DECLARE X K L)
              (IF (NOT $ONFLOOR)
                  THEN
                  (FAIL))
              (EXISTS (CONNECTS ←K ←L $M))
              (GOAL $GO (INROOM ROBOT $L))
              (GOAL $GO (NEXTTO ROBOT $K))
              (MAPC (QUOTE (TUPLE (ATROBOT ←X)
                                  (NEXTTO ROBOT ←X)))
                    $DELETE)
              (ASSERT (INROOM ROBOT $M))
              ($BUILD (' (:$GOTHRUDOORACTION $K]
```

THE GOTO2 OPERATOR

```
[RPAQQ GOTO2 (LAMBDA (NEXTTO ROBOT ←M)
        (PROG (DECLARE X Y)
              (IF (NOT $ONFLOOR)
                  THEN
                  (FAIL))
              (ATTEMPT (EXISTS (INROOM $M ←X))
                       (GOAL $GO (INROOM ROBOT $X))
                       THEN
                       (GO FINISH)
                       ELSE
                       (GOAL $GO (INROOM ROBOT ←X))
                       (EXISTS (CONNECTS $M $X ←Y)))
         FINISH
              (MAPC (QUOTE (TUPLE (ATROBOT ←X)
                                  (NEXTTO ROBOT ←X)))
                    $DELETE)
              (ASSERT (NEXTTO ROBOT $M))
              ($BUILD (' (:$GOTO2ACTION $M]
```

THE FUNCTION DELETE

```
[RPAQQ DELETE (LAMBDA ←EXP
        (PROG (DECLARE X)
              (ATTEMPT (SETQ ←X (EXISTS $EXP))
                       THEN
                       (DENY $X]
```

THE GOTO1 OPERATOR

```
[RPAQQ GOTO1 (LAMBDA (ATROBOT ←M)
        (PROG (DECLARE X)
              (IF (NOT $ONFLOOR)
                  THEN
                  (FAIL))
              (EXISTS (LOCINROOM $M ←X))
              (GOAL $GO (INROOM ROBOT $X))
              (MAPC (QUOTE (TUPLE (ATROBOT ←X)
                                  (NEXTTO ROBOT ←X)))
                    $DELETE)
              (ASSERT (ATROBOT $M))
              ($BUILD (' (:$GOTO1ACTION $M]
```

THE PUSHTO OPERATOR

```
[RPAQQ PUSHTO (LAMBDA (NEXTTO ←M ←N)
        (PROG (DECLARE X)
              (IF (NOT $ONFLOOR)
                  THEN
                  (FAIL))
              (EXISTS (PUSHABLE $M))
              (ATTEMPT (EXISTS (INROOM $M ←X))
                       (EXISTS (INROOM $N $X))
                       ELSE
                       (EXISTS (INROOM $M ←X))
                       (EXISTS (CONNECTS $N $X ←Y)))
              (GOAL $GO (NEXTTO ROBOT $M))
              (MAPC (QUOTE (TUPLE (ATROBOT ←X)
                                  (AT $M ←X)
                                  (NEXTTO ROBOT ←X)
                                  (NEXTTO $M ←X)
                                  (NEXTTO $X $M)))
                    $DELETE)
              (ASSERT (NEXTTO $M $N))
              (ASSERT (NEXTTO $N $M))
              (ASSERT (NEXTTO ROBOT $M))
              ($BUILD (' (:$PUSHTOACTION (TUPLE $M $N]
```

THE CLIMBONBOX OPERATOR

```
[RPAQQ CLIMBONBOX (LAMBDA (ON ROBOT ←M)
         (PROG (DECLARE X)
               (IF (NOT $ONFLOOR)
                   THEN
                   ($CLIMBOFFBOX))
               (EXISTS (TYPE $M BOX))
               (GOAL $GO (NEXTTO ROBOT $M))
               ($DELETE (QUOTE (ATROBOT ←X)))
               (SETQ ←ONFLOOR FALSE)
               (ASSERT (ON ROBOT $M))
               ($BUILD (' (:$CLIMBONBOXACTION $M]
```

THE CLIMBOFFBOX OPERATOR

```
[RPAQQ CLIMBOFFBOX (LAMBDA (TUPLE)
         (PROG (DECLARE M)
               (EXISTS (ON ROBOT ←M))
               (EXISTS (TYPE $M BOX))
               ($DELETE (QUOTE (ON ROBOT $M)))
               (SETQ ←ONFLOOR TRUE)
               ($BUILD (' (:$CLIMBOFFBOXACTION $M]
```

THE TURNONLIGHT OPERATOR

```
[RPAQQ TURNONLIGHT (LAMBDA (STATUS ←M ON)
         (PROG (DECLARE N)
               (EXISTS (TYPE $M LIGHTSWITCH))
               (EXISTS (TYPE ←N BOX))
               (GOAL $DO (NEXTTO $N $M))
               (GOAL $DO (ON ROBOT BOX1))
               ($DELETE (QUOTE (STATUS $M OFF)))
               (ASSERT (STATUS $M ON))
               ($BUILD (' (:$TURNONLIGHTACTION $M]
```

THE FUNCTION BUILD

```
[RPAQQ BUILD (LAMBDA ←X
         (SETQ ←ANSWER (CONS $X $ANSWER]
```

THE FUNCTION SOLVE

```
[RPAQQ SOLVE (LAMBDA ←PROBLEM
         (PROG (DECLARE X)
               (SETQ ←X (REVERSE $PROBLEM))
               (RETURN (PROG (DECLARE)
                             $$X]
```

The following is a model of the robot world. Expressions are evaluated by the QA4 evaluator ("!") and stored in the net.

```
(DEFLIST(QUOTE(
  [SETUP ([(! (SETQ ←DO (TUPLE CLIMBONBOX TURNONLIGHT PUSHTO]
          [(! (SETQ ←GO (TUPLE GOTHRUDOOR GOTO1 GOTO2]
          ((! (SETQ ←ONFLOOR TRUE)))
          [(! (SETQ ←ANSWER (' (TUPLE]
          [(! (ASSERT (INROOM LIGHTSWITCH1 ROOM1]
          [(! (ASSERT (INROOM ROBOT ROOM1]
          [(! (ASSERT (ATROBOT E]
          [(! (ASSERT (LOCINROOM F ROOM4]
          [(! (ASSERT (PUSHABLE BOX1]
          [(! (ASSERT (PUSHABLE BOX2]
          [(! (ASSERT (PUSHABLE BOX3]
          [(! (ASSERT (INROOM BOX1 ROOM1]
          [(! (ASSERT (INROOM BOX2 ROOM1]
          [(! (ASSERT (INROOM BOX3 ROOM1]
          [(! (ASSERT (STATUS LIGHTSWITCH1 OFF]
          [(! (ASSERT (TYPE LIGHTSWITCH1 LIGHTSWITCH]
          [(! (ASSERT (TYPE BOX1 BOX]
          [(! (ASSERT (TYPE BOX2 BOX]
          [(! (ASSERT (TYPE BOX3 BOX]
          [(! (ASSERT (AT LIGHTSWITCH1 D]
          [(! (ASSERT (AT BOX1 A]
          [(! (ASSERT (AT BOX2 B]
          [(! (ASSERT (AT BOX3 C]
          [(! (ASSERT (CONNECTS DOOR1 ROOM1 ROOM5]
          [(! (ASSERT (CONNECTS DOOR1 ROOM5 ROOM1]
          [(! (ASSERT (CONNECTS DOOR2 ROOM2 ROOM5]
          [(! (ASSERT (CONNECTS DOOR2 ROOM5 ROOM2]
          [(! (ASSERT (CONNECTS DOOR3 ROOM3 ROOM5]
          [(! (ASSERT (CONNECTS DOOR3 ROOM5 ROOM3]
          [(! (ASSERT (CONNECTS DOOR4 ROOM4 ROOM5]
          ((! (ASSERT (CONNECTS DOOR4 ROOM5 ROOM4]
```

THE 3 INITIAL PROBLEM STATEMENTS

```
[3BOXPROBLEM (((! ($SOLVE (LIST (GOAL $DO (NEXTTO BOX1 BOX2))
                                (GOAL $DO (NEXTTO BOX2 BOX3]
[GOROOMPROBLEM (((! ($SOLVE (GOAL $GO (ATROBOT F]
[TURNONLIGHTPROBLEM (((! ($SOLVE (GOAL $DO (STATUS LIGHTSWITCH1
 ON] ))(QUOTE HISTORY))
```

TRACE OF THE SOLUTION OF THE PROBLEM OF TURNING ON A LIGHT

```
1   (GOAL $DO (STATUS (TUPLE LIGHTSWITCH1 ON)))
2   GOALCLASS ← (TUPLE CLIMBONBOX TURNONLIGHT PUSHTO)
3   FAILURE
4     LAMBDA TURNONLIGHT
5     M ← LIGHTSWITCH1
6     (EXISTS (TYPE (TUPLE $M LIGHTSWITCH)))
7     (EXISTS (TYPE (TUPLE ←N BOX)))
8     N ← BOX1
9     (GOAL $DO (NEXTTO (TUPLE $N $M)))
10    GOALCLASS ← (TUPLE CLIMBONBOX TURNONLIGHT PUSHTO)
11    FAILURE
12      LAMBDA PUSHTO
13      N ← LIGHTSWITCH1
14      M ← BOX1
15      (EXISTS (PUSHABLE $M))
16      (EXISTS (INROOM (TUPLE $M ←X)))
17      X ← ROOM1
18      (EXISTS (INROOM (TUPLE $N $X)))
19      (GOAL $GO (NEXTTO (TUPLE ROBOT $M)))
20      GOALCLASS ← (TUPLE GOTHRUDOOR GOTO1 GOTO2)
21      FAILURE
22        LAMBDA GOTO2
23        M ← BOX1
24        (EXISTS (INROOM (TUPLE $M ←X)))
25        X ← ROOM1
26        (GOAL $GO (INROOM (TUPLE ROBOT $X)))
27        GOALCLASS ← (TUPLE GOTHRUDOOR GOTO1 GOTO2)
28          LAMBDA DELETE
29          EXP ← (ATROBOT ←X)
30          (EXISTS $EXP)
31          X ← E
32          X ← (ATROBOT E)
33          (DENY $X)
34          LAMBDA DELETE
35          EXP ← (NEXTTO (TUPLE ROBOT ←X))
36          (EXISTS $EXP)
37          FAILURE
38        (ASSERT (NEXTTO (TUPLE ROBOT $M)))
39          LAMBDA BUILD
40          X ← ($GOTO2ACTION BOX1)
41          ANSWER ← (TUPLE ($GOTO2ACTION BOX1))
42        LAMBDA DELETE
43        EXP ← (ATROBOT ←X)
44        (EXISTS $EXP)
45        FAILURE
46        LAMBDA DELETE
47        EXP ← (AT (TUPLE $M ←X))
48        (EXISTS $EXP)
49        FAILURE
50        LAMBDA DELETE
```

```
51          EXP ← (NEXTTO (TUPLE ROBOT ←X))
52          (EXISTS $EXP)
53          X ← BOX1
54          X ← (NEXTTO (TUPLE ROBOT BOX1))
55          (DENY $X)
56          LAMBDA DELETE
57          EXP ← (NEXTTO (TUPLE $M ←X))
58          (EXISTS $EXP)
59          FAILURE
60          LAMBDA DELETE
61          EXP ← (NEXTTO (TUPLE $X $M))
62          (EXISTS $EXP)
63          FAILURE
64        (ASSERT (NEXTTO (TUPLE $M $N)))
65        (ASSERT (NEXTTO (TUPLE $N $M)))
66        (ASSERT (NEXTTO (TUPLE ROBOT $M)))
67          LAMBDA BUILD
68          X ← ($PUSHTOACTION (TUPLE BOX1 LIGHTSWITCH1))
69          ANSWER ← (TUPLE ($PUSHTOACTION (TUPLE BOX1
LIGHTSWITCH1)) ($GOTO2ACTION BOX1))
70    (GOAL $DO (ON (TUPLE ROBOT BOX1)))
71    GOALCLASS ← (TUPLE CLIMBONBOX TURNONLIGHT PUSHTO)
72    FAILURE
73      LAMBDA CLIMBONBOX
74      M ← BOX1
75      (EXISTS (TYPE (TUPLE $M BOX)))
76      (GOAL $GO (NEXTTO (TUPLE ROBOT $M)))
77      GOALCLASS ← (TUPLE GOTHRUDOOR GOTO1 GOTO2)
78        LAMBDA DELETE
79        EXP ← (ATROBOT ←X)
80        (EXISTS $EXP)
81        FAILURE
82      ONFLOOR ← FALSE
83      (ASSERT (ON (TUPLE ROBOT $M)))
84        LAMBDA BUILD
85        X ← ($CLIMBONBOXACTION BOX1)
86        ANSWER ← (TUPLE ($CLIMBONBOXACTION BOX1) (
$PUSHTOACTION (TUPLE BOX1 LIGHTSWITCH1)) ($GOTO2ACTION
BOX1))
87      LAMBDA DELETE
88      EXP ← (STATUS (TUPLE $M OFF))
89      (EXISTS $EXP)
90      FAILURE
91    (ASSERT (STATUS (TUPLE $M ON)))
92      LAMBDA BUILD
93      X ← ($TURNONLIGHTACTION LIGHTSWITCH1)
94      ANSWER ← (TUPLE ($TURNONLIGHTACTION LIGHTSWITCH1)
($CLIMBONBOXACTION BOX1) ($PUSHTOACTION (TUPLE BOX1
LIGHTSWITCH1)) ($GOTO2ACTION BOX1))
95    LAMBDA SOLVE
```

```
96   PROBLEM ← (TUPLE ($TURNONLIGHTACTION LIGHTSWITCH1)
($CLIMBONBOXACTION BOX1) ($PUSHTOACTION (TUPLE BOX1
LIGHTSWITCH1)) ($GOTO2ACTION BOX1))
97   X ← (TUPLE ($GOTO2ACTION BOX1) ($PUSHTOACTION (TUPLE
BOX1 LIGHTSWITCH1)) ($CLIMBONBOXACTION BOX1) (
$TURNONLIGHTACTION LIGHTSWITCH1))
98   (RETURN (PROG (DECLARE) $$X))
99 (PROG (DECLARE) ($GOTO2ACTION BOX1) ($PUSHTOACTION
(TUPLE BOX1 LIGHTSWITCH1)) ($CLIMBONBOXACTION BOX1) (
$TURNONLIGHTACTION LIGHTSWITCH1))
```

THE ANSWER RETURNED BY QA4

```
(PROG (DECLARE) ($GOTO2ACTION BOX1) ($PUSHTOACTION (TUPLE BOX1
LIGHTSWITCH1)) ($CLIMBONBOXACTION BOX1) ($TURNONLIGHTACTION
LIGHTSWITCH1))
```

Appendix II

THE DISCRIMINATION NET

I BACKGROUND

A. Canonical Forms

Within QA4, the requirement that expressions have a standard form is the result of a more basic need. QA4 programs operate mainly through manipulating expressions by:

- Constructing new expressions
- Decomposing with pattern matching
- Associatively indexing with patterns
- Retrieving properties of expressions.

The EXISTS statement is an example of associatively indexing. It requires that all the expressions in the system that could match a pattern be examined. It is this process together with the problem of retrieving properties of expressions that necessitates a canonical form approach for all expression storage. For example, suppose a program constructs a tuple, say (TUPLE 1 2 3), and must associate a property, say property-value TRUE under property-name CONTAINS-PRIME, with the tuple. Later, if another program constructed the same tuple, it must have access to the property CONTAINS-PRIME. With only tuples, this is an easy task. When sets and bags are added to the language, the task is only slightly more complicated. There are, however, secondary benefits. When sets are kept in a standard internal order, all the set manipulation primitive functions such as UNION can operate as one-pass iterative programs. A standard order for sets also

simplies pattern matching. When expressions with bound variables are added to a language that contains sets, the task of finding a standard form becomes complicated and a heuristic approach for the retrieval system appears best.

A straightforward approach for associating properties with expression is to extend the ATOM feature of LISP. Each expression will be represented by a single pointer to a property-list, and one item of that property-list will be the syntactic form of the expression. In LISP an atom, say ABC, is represented by a pointer to a property-list, and one item of the property-list (usually PNAME) contains the internal representation for ABC. Moreover, every occurrence of ABC in the system is represented by the same pointer. In QA4, each expression is represented by a pointer to a property-list, and the property EXPV is the syntactic form of the expression. Thus the QA4 expression (TUPLE 1 2 3) is a list of the form

(NETEXPRESSION EXPV (TUPLE 1 2 3) 0 (0 CONTAINS-PRIME TRUE)) .

The first item, NETEXPRESSION, identifies the list and is used as a temporary marker during garbage collection.

Normally all properties of a QA4 expression are stored with the context mechanism. CONTAINS-PRIME in the above example illustrates this form of storage. However, a few properties, called syntactic properties, are distinguished and may never be changed. They are not stored with the context mechanism and their property-names are on the

top level of the list representing the expression. In our example, EXPV is a syntactic property, and CONTAINS-PRIME is a normal property that has the value TRUE under context ((0) 0).

B. Basic Mechanisms

Since each occurrence of an expression that appears the same must, in fact, be the same, the expression constructors are the core of any system that implements an expression property-association feature. For example, given the list (1 2 3), the tuple constructor must find the pointer to the expression

(NETEXPRESSION EXPV (TUPLE 1 2 3) ...) .

For sets, a preliminary reordering is required. Thus if the QA4 set constructor is given the elements (3 2 1) it would find the expression

(NETEXPRESSION EXPV (SET 1 2 3) ...) .

If bound variables occur in the expression, they must be converted to standard names before the lookup can be accomplished. For example:

(LAMBDA (TUPLE ←X ←Y) ($F $X $Y))

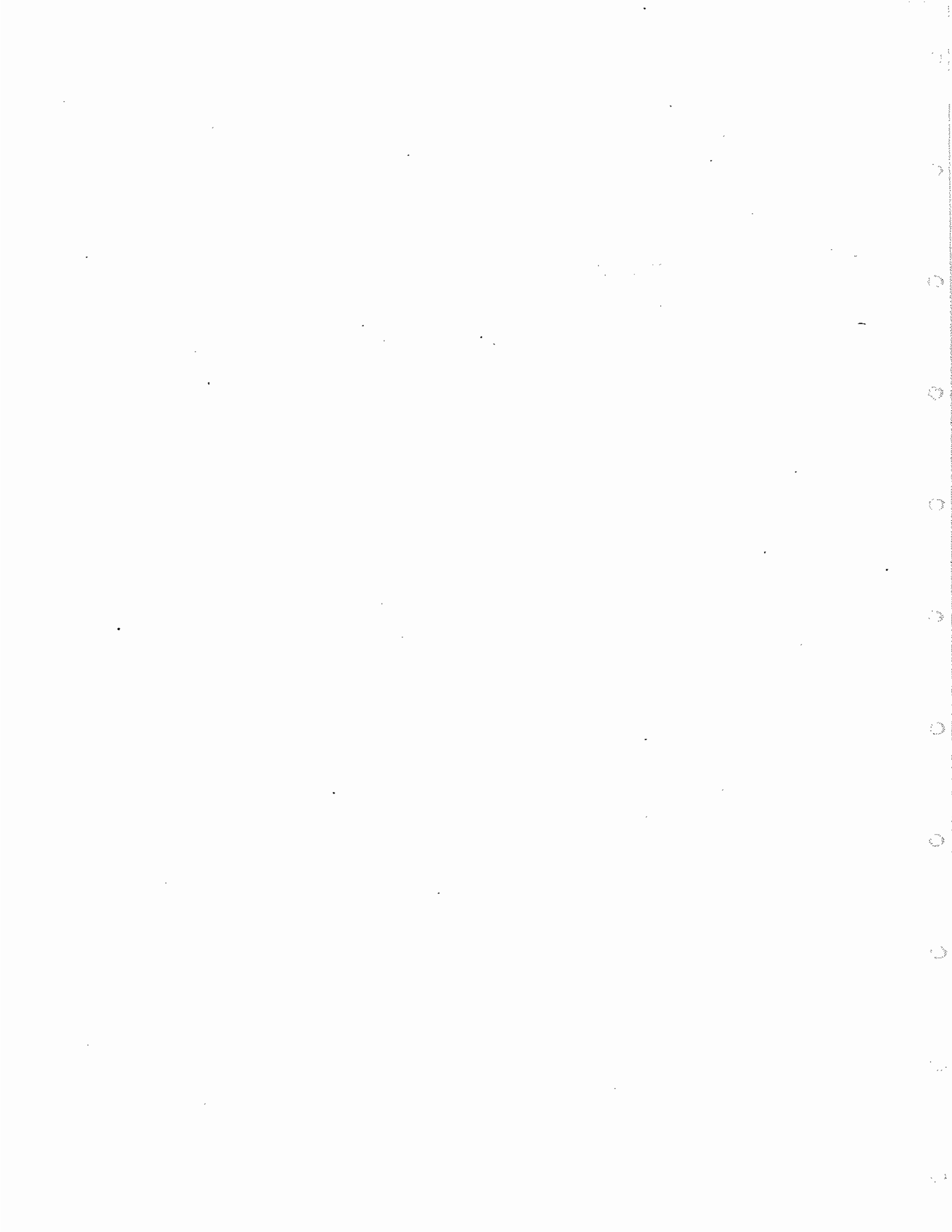
could be converted to

(LAMBDA (TUPLE ←V1 ←V2) ($F $V1 $V2))

and then the lookup performed. Then if the expression

(LAMBDA (TUPLE ←Y ←X) ($F $Y $X))

were to be constructed, Y would be replaced by V1 and X by V2. Now, since the standard form would represent both expressions, properties associated with either would be available just as in the tuple example.



II FIXED RETRIEVAL, REORDERING, and RENAMING

A. Coordinate Indexing

The requirement that the data must be associatively retrievable combined with the lookup requirements of the constructors demands a data structure where items are indexed by their subexpressions. PLANNER (Hewitt, 1972) uses a scheme called coordinate indexing. The coordinates of an expression are taken from the natural numbering of the positions of subexpressions. For example, in the expression (A (B C)(D (E F) G)), the subexpressions have the following coordinates:

Coordinate	Subexpression
1	A
2	(B C)
3	(D (E F) G)
(2, 1)	B
(2, 2)	C
(3, 1)	D
(3, 2)	(E F)
(3, 3)	G
(3, 2, 1)	E
(3, 2, 2)	F

A hash table is generated for each coordinate and the items of the hash table are all the expressions that have occurred in that coordinate position in some expression. Associated with each hashed item is a list of the expressions in which the hashed expression occurred. Suppose we had the three expressions

e1 (NETEXPRESSION EXPV (TUPLE A B) ...)

e2 (NETEXPRESSION EXPV (TUPLE B e1) ...)

e3 (NETEXPRESSION EXPV (TUPLE A D) ...) .

The user forms printed for e1, e2, e3 would be (TUPLE A B), (TUPLE B(TUPLE A B)), and (TUPLE A D), respectively. The coordinate hash tables for these three expressions would be

<u>Coordinate 1</u>		<u>Coordinate 2</u>	
A	(e1 e3)	B	(e1)
B	(e2)	(TUPLE A B)	(e2)
		D	(e3)

The task for a constructor is now made quite easy. Given a collection of components for an expression, say e1, e2, ... en, the constructor first looks for e1 in hash table 1 and either finds a list or nothing. If it finds nothing, it is constructing a new expression and must enter each e and its subexpression in the hash tables. If it finds a list, say ℓ, it then looks for e2 in the table for coordinate 2. If it finds a list, it intersects the list with ℓ and makes a new ℓ. If the new ℓ is empty or if it found nothing, it enters the new expression as before. If the new ℓ is not empty, it continues the intersection process.

To associatively index, essentially the same process can be carried out, but the variable items are skipped. The ℓ resulting from all the intersections is then reduced further by forcing the variables to correspond properly. This final ℓ is the result of the search. During associative searching, the tables are used in a depth-first left-right

manner rather than just on the top level, and only constant expressions (ones without variables) can be used. For example, to retrieve

(TUPLE A (TUPLE ←X B))

we would only check coordinate tables 1 and (2, 2).

This indexing scheme has the advantage that the retrieval time remains almost constant.

B. Reordering

To use any storage method, the expressions must first be reordered and variables must be renamed. For all the parts of an expression that have order, such as tuples and applications, nothing needs to be done.

For sets, the reordering task can be accomplished by assigning each expression that occurs in a set an index number and then ordering the items of sets based on their index numbers. Since each expression given to the set constructor is an expression with a property-list the index numbers can be kept as syntactic properties on the property-lists. For example, to construct the set from the components (TUPLE 1 2) (TUPLE 2 1), 3, and A, the 3 would become the first item because numbers are always sorted to the front and then sorted in their natural order. The other three items would each have an index number. We may have

e1 (NETEXPRESSION EXPV (TUPLE 1 2) INDEX 3 ...)

e2 (NETEXPRESSION EXPV (TUPLE 2 1) INDEX 2 ...)

(atom-flag PNAME "A" INDEX 7) .

Thus the set constructor would return the expression

```
(NETEXPRESSION EXPV (SET 3 e2 e1 A) ...)   .
```

C. Bound Variable Renaming

Bound variable renaming, however, is a more difficult task--so difficult it appears to prohibit the use of fixed retrieval methods such as coordinate indexing. In QA4, only the syntactic forms SET and BAG are unordered. Thus if we scan a QA4 expression in a depth-first left-right manner, an order is naturally imposed on all variables except those that only occur in sets and bags. For example,

```
(LAMBDA (SET ←X ←Y) (TUPLE ←Y ←X))
```

naturally orders Y before X because of the final tuple.

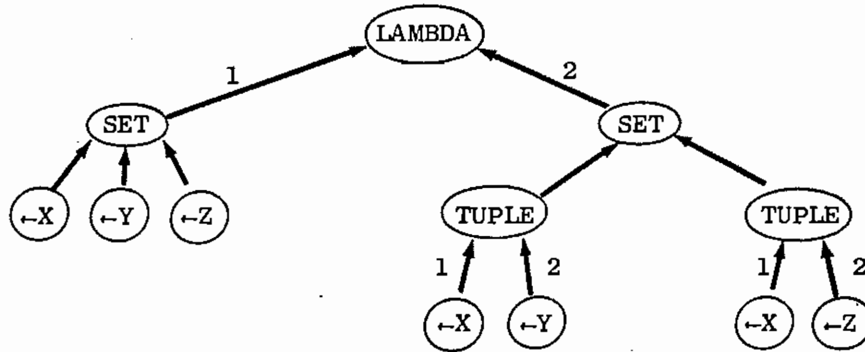
Sometimes, however, the order is only partial and at other times apparent order is not really present. For example, in

```
(LAMBDA (SET ←X ←Y) (SET (TUPLE 1 ←X) (TUPLE 2 ←Y)))
```

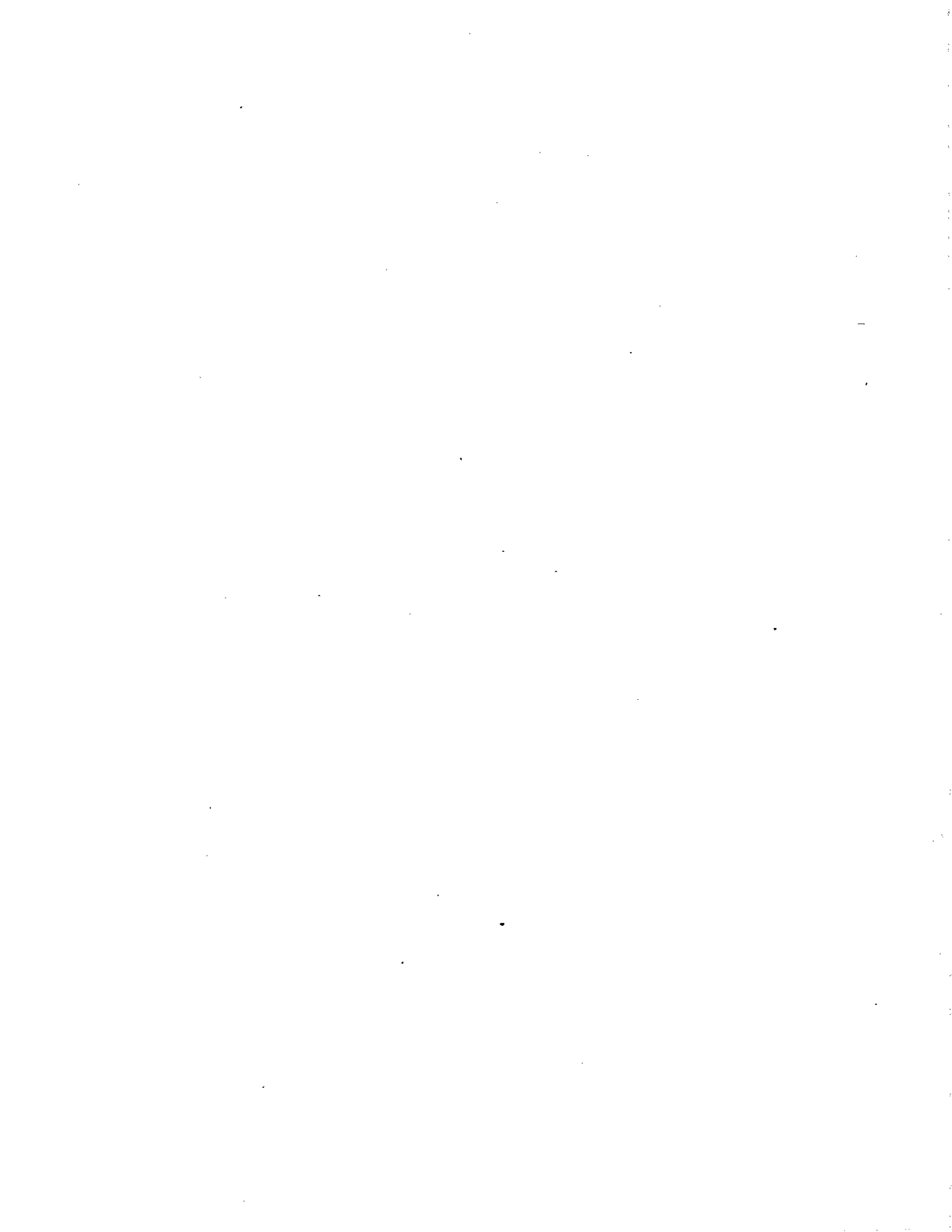
X and Y are not ordered, while in

```
(LAMBDA (SET ←X ←Y ←Z) (SET (TUPLE ←X ←Y) (TUPLE ←X ←Z)))
```

X is less than Y or Z, but Y and Z are not ordered. To see this order, construct a graph whose nodes are the expressions at each coordinate and whose directed arcs connect the coordinates appropriately and are labeled with the natural position numbers if they exist. Thus the last example would generate the graph:



Some arcs do not have numbers and thus there are variables that do not have a path completely numbered up to the head of the tree. For variables that do have a numbered path, we can easily assign an order. For the other variables, we can assign a partial ordering by first ordering the nodes they point to and then use the frequency of their position numbers within that grouping. This process can be carried on iteratively, eliminating variables from consideration until only truly unorderable variables remain. The process, however, is complicated and cumbersome.



III HEURISTIC RETRIEVAL WITHOUT RENAMING

A. The Heuristic Technique

The heuristic approach used in QA4 eliminates the job of renaming while maintaining competitive retrieval times and storage growth. We have implemented a retrieval method that, given a syntactic form, retrieves a list of candidates (some of which may be incorrect), that must be checked by the pattern matcher. That is, given the EXPV property of an expression, the retrieval system returns a collection of property-list expressions, one of which may have an EXPV property that matches the input expression.

For example, suppose our system contains the expressions

e1 (NETEXPRESSION EXPV (SET 1 1) ...)

e2 (NETEXPRESSION EXPV (SET 1 2) ...)

e3 (NETEXPRESSION EXPV (TUPLE 1 2) ...)

and we ask to retrieve (SET 1 ←X). The system would return the list (e1 e2), and then use the pattern matcher to verify that each expression does in fact match. If we ask to retrieve (SET ←X ←X) the system would still return (e1 e2), and the pattern matcher would be required to verify that e1 did match while e2 did not.

The system avoids renaming variables during construction because it keeps original variables and always checks to see if the expression already exists or if one exists that could match it. For example, if we constructed the expression

(LAMBDA (BAG ←X ←Y) (PLUS $X $Y))

and later attempted to construct

(LAMBDA (BAG ←U ←V) (PLUS $U $V))

the LAMBDA constructor would find that the expression to be constructed would match the previous expression and thus would return the property-list form of the previous expression as the constructed form of the expression containing U and V. In order to use the pattern matcher for this process, it must be inhibited so that variables must match variables. Normally, a variable in a pattern can match any subexpression. For example, (SET ←X ←Y) matches (SET 1 (SET 2 3)), but when checking for syntactic identity during construction variables can only match variables.

B. The Discrimination Net

The retrieval system uses a discrimination net. Nodes of the net are either terminal or nonterminal. Nonterminal nodes have two components: a coordinate with an optional type-flag and a list of branches. The branches are pairs of features and corresponding subnodes. For example, the expressions

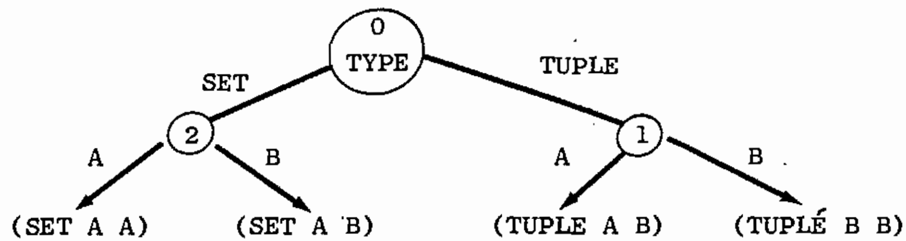
(SET A A)

(SET A B)

(TUPLE A B)

(TUPLE B B)

could be stored in the tree.

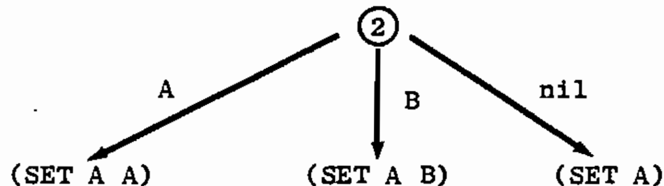


The top node has coordinate 0 and the optional type-flag, so it means that the branches emanating from it are based on the syntactic type of the top level of the expression, sets are in the branch to the left, and tuples are in the branch to the right. The node with coordinate 2 means that its branches are based on that coordinate of the expression. Terminal nodes have a single property-list form expression on them.

C. Construction Search

For construction, the search algorithm is quite simple. Given a syntactic form, we fetch the feature specified by the top node, choose the appropriate branch, and continue the process until we reach a terminal node. At that point we have either found the property-list for the expression or we must add a new expression to the net by replacing the terminal node with a new nonterminal node that has as branches the old terminal node and a new terminal node that contains the expression we just constructed. The coordinate for this new nonterminal node is constructed automatically by searching for the first difference in the two expressions, except that a variable can never be used as a coordinate.

Sometimes a coordinate at a node does not apply to an expression. For example, if we searched the above net with (SET A), the coordinate 2 would not apply. In this case, we build a default branch and continue in the normal way. The left side of the above tree would become:



D. Associative Search

During associative retrieval, a slightly more complicated search algorithm is used. If we are searching with a pattern and a coordinate specifies a variable in the pattern or the search for the coordinate passes through a set or bag, the result of the search is the union of the search applied to all the branches at that node. Thus if we searched the previous tree for the expression (SET A ←X), when we were at the node for coordinate 2, the search algorithm would be called recursively with the nodes emanating from both branches A and B.

If the coordinate specifies a constant, however, and there is no branch corresponding to that constant, then the answer for that branch is nil. It is this last search modification that requires that only atoms be used as features. Otherwise, if any expression could be used to identify branches it would be time-consuming to continually check if a complicated expression contained a variable.

IV SUMMARY

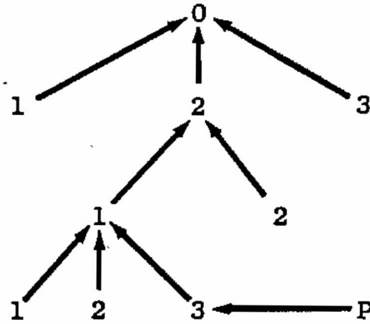
A. Storage Consumption

If the net grows in an even fashion, the branching at each node would be the same, say it is B. Then, if the net has T terminal nodes, the depth of the tree would be $\log_B T$. Let $N = \log_B T - 1$, then N is the number of nonterminal nodes in a path of the net; that is, N is the depth of the net. The total number of nonterminal nodes is now the sum of the geometric series

$$S = 1 + B + B^2 + \ldots + B^N = \frac{B^{N+1} - 1}{B - 1} = \frac{T - 1}{B - 1} \quad .$$

Each nonterminal node would have a coordinate and a collection of branches. Thus S, the overhead storage for the net, would be $K * (T - 1)/(B - 1)$, and would grow linearly with the number of terminal nodes. Currently, K is approximately 25, an excessive amount. The net is kept in a property-list format, and statistics are kept on every reference to every property. This is done to study the evenness of growth and the regularity of the search paths. K could, however, be made quite small. In fact, it could be reduced to $K = 1 + 3B$. The single cell could point to a terminal node coordinate tree that would be used for the entire net.

For example, the trees



can be kept as a list structure, and P indicates coordinate (2 1 3); 3B is the storage necessary for a list of branches.

B. Time

It is difficult to even estimate the theoretical ratio for retrieval times between alternative storage systems. Real timing, on the other hand, may be more a function of the operating system than the storage system. Thus we have no meaningful comparisons.

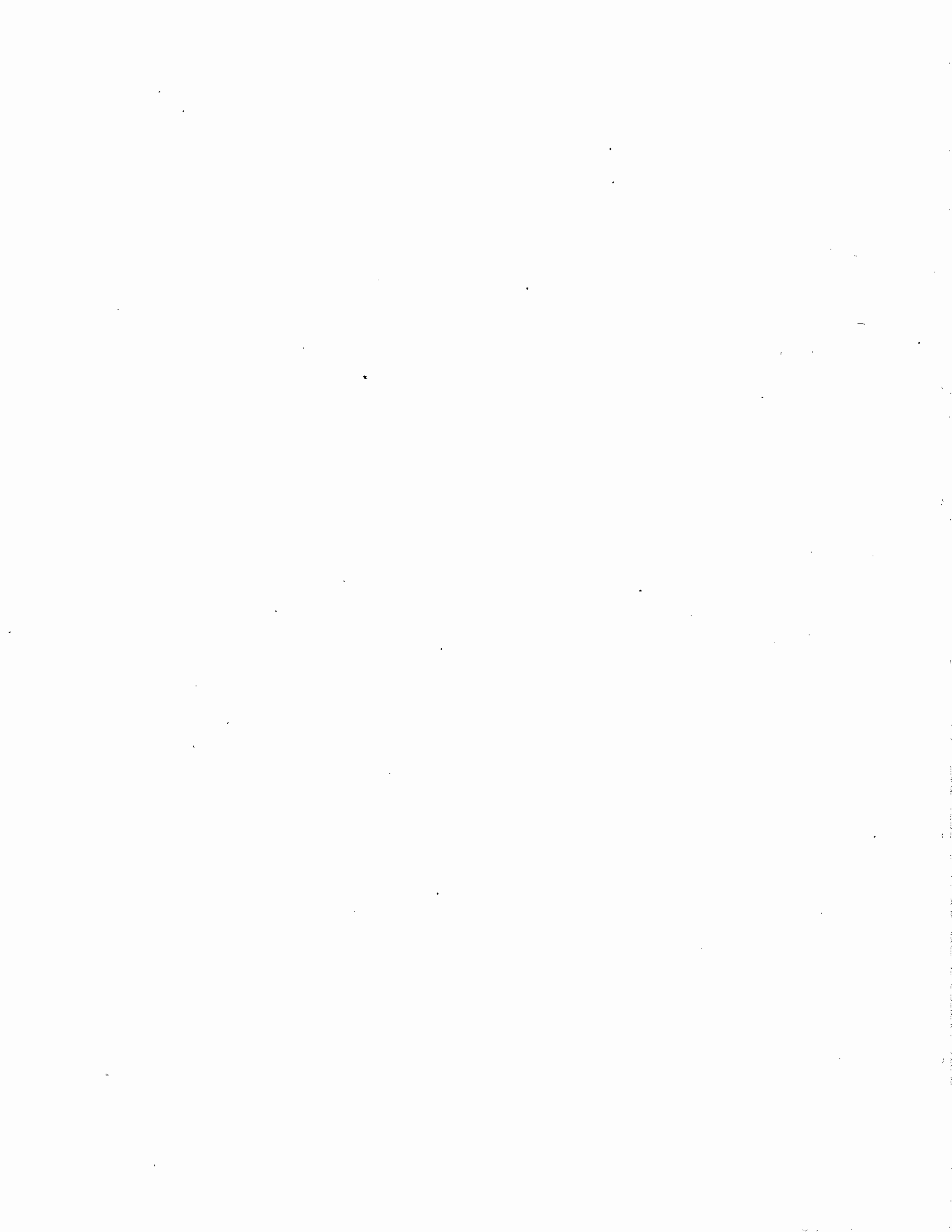
C. Disadvantages

The major disadvantage with the discrimination net is not with the net system itself but with the way the QA4 interpreter uses it. Within QA4, all expressions must have a canonical form. Thus all programs are kept in the net. Moreover, during evaluation, all intermediate results are made into expressions and stored in the net. This permits a clear, unified semantics for the QA4 language. For example, the definition of EQUAL can be that the set it applies to has a single element after evaluation. But, this excessive use of the net generates a sizable

amount of garbage. Thus, on the one hand, we have a language with unusual but extraordinarily useful semantics, while on the other hand, it eats up storage too fast. We hope, however, that the problem can be solved by a better interpreter organization. In the meantime, the net appears to offer the best solution for canonical construction and associative retrieval.

Appendix III

CONTEXTS



I BINDINGS

A. Properties

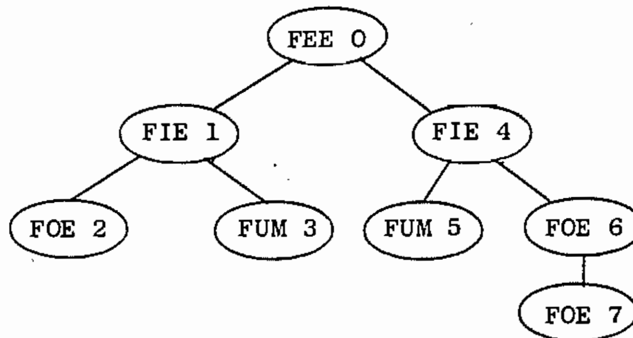
Properties of QA4 expressions are distinguished along three dimensions. The first is the name of the property; EXPV and NETVALUE are common names. The second parameter of the GET statement is a name. For example:

(GET (ON $X $Y) TIME)

retrieves the value of the property TIME from the instantiated form of the expression (ON $X $Y), and TIME is the name. All of the names used in the system form a dimension with discrete coordinates. Thus, for each expression we may visualize a space whose first dimension is the enumeration of names used in the system.

B. Dynamic Context

The second dimension is the binding information. Suppose we have subroutines FEE, FIE, FOE, and FUM. We call FEE, and the following takes place:



meaning that

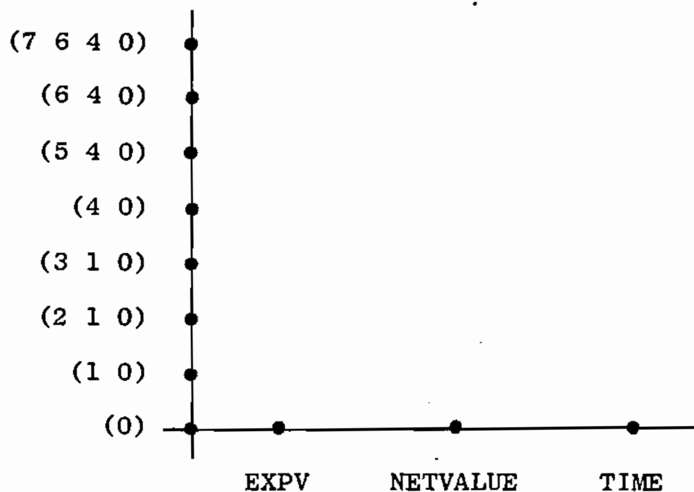
```
FEE calls
     FIE calls
          FOE returns
        calls
          FUM returns
        returns
     calls
       FIE calls
          FUM returns
        calls
          FOE calls
               FOE returns
              returns
        returns
     returns   .
```

The numbers in the nodes represent the order of subroutine calls. At any instant during the execution of the program only one path of the tree is active. Paths to the left of it have already been executed, and paths to the right are yet to be executed. Thus, for each incarnation of each subroutine there is a path from the bottom node to the root of the tree that contains the binding information for that particular

call on that subroutine. For example, (2 1 0) and (7 6 4 0) are paths for two different incarnations of FOE. All the paths, subpaths included, make the second dimension by which properties are accessed.

The paths are called "contexts." By associating a context with each incarnation of a subroutine we may fetch and store properties within a "dispersed state" system and yet have the same apparent bindings provided by stack organized systems. This dynamic binding is accomplished by always looking for the most relevant context first. That is, looking for a binding closer to the head of the path (e.g., 7) before looking for more global bindings (e.g., 5).

We may now visualize the space for each expression with two dimensions. For our example we would have



Each indicator is a coordinate of the horizontal dimension, and each context is a coordinate of the vertical dimension. Each expression has such a space and may thus have a different value for each property

under each context. In practice, however, this is rarely the case. Usually only a few points of the space have values different from adjacent points. In a programming sense, bindings do not change very often. So instead of copying values to each new context coordinate, we modify the retrieval mechanism. Given a context, say (7 6 4 0), and an indicator, say TIME, the retrieval program scans down selected points of the line defined by the indicator coordinate. That is, it first checks (TIME,(7 6 4 0)) for a value, if none exists it checks (TIME,(6 4 0)), then (TIME,(4 0)) and finally (TIME,(0)). The scan algorithm takes advantage of the tree structure of the context coordinate.

C. Backtracking Context

The third dimension is similar to the second but provides for backtracking. Suppose the system is in a binding context C and must establish a context C' such that (1) all properties of expressions are carried forward from C to C', (2) all property manipulation takes place with respect to C', and (3) all the values with respect to C can be restored. This can be established by creating a backtracking context tree similar to the dynamic binding context tree. Nodes in this tree are generated when a backtracking point is established.

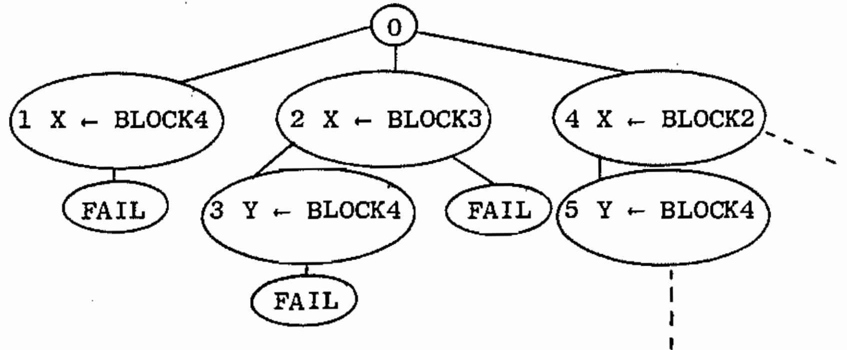
For example, suppose there was a stack of blocks: BLOCK4, BLOCK3, BLOCK2, BLOCK1; where only BLOCK3 is a cube; BLOCK4 is on BLOCK3, BLOCK2 and BLOCK1; and BLOCK2 is on BLOCK1. The statements

(1) (EXISTS (ON BLOCK1 ←X))

(2) (EXISTS (UNDER $X ←Y))

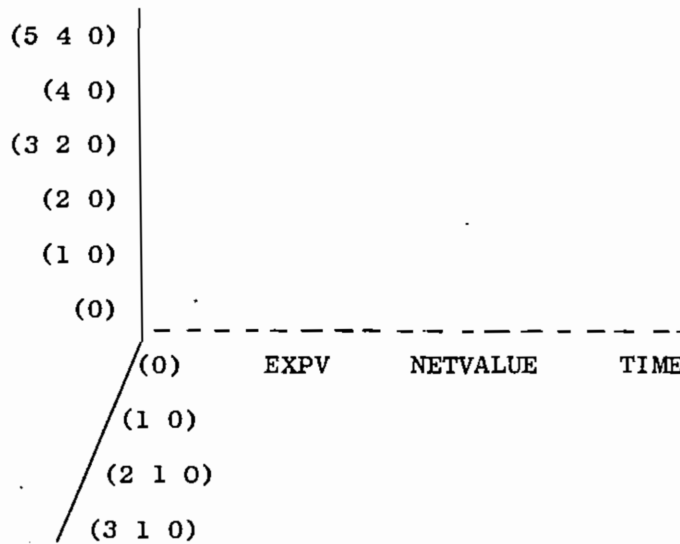
(3) (IF (NOT (GET (CUBE $Y) NETVALUE)) THEN FAIL)

could form the tree



Both (1 0) and (5 4 0) are backtracking contexts. Just like dynamic contexts, only one path is active at any instant, paths to the left have been executed and paths to the right are yet to be executed.

We may now visualize the space for each expression with the full three dimensions. For our example we would have:



Creating a new backtracking context creates a new plane in the space. All values are projected into the new plane and any changes are made only in the new plane. When a subroutine returns, any changes made to variables global to the subroutine must be kept. Thus if a subroutine, S, has dynamic context D, and backtracking context B, and called subroutine S', S' would begin with dynamic context D' (derived from D) and backtracking context B. When S' returns, however, backtracking points may have been established and the backtracking context may be B'. Thus when S' returns to S, S retains its original dynamic context D, but adopts the new backtracking context B'. These three dimensions could be searched for values in any order, so efficiency considerations guided the algorithms actually implemented.

D. Benefits of Dispensed State

By using these paths as the coordinates of the three dimensions, we have a system that can backtrack and still have a dispersed state. By keeping the dimensions independent, QA4 provides a wide range of binding and backtracking options. The examples in the next section on process structures illustrate the many combinations of binding and backtracking available in QA4.

By keeping all three dimensions free from any control information, backtracking is simplified and experiments such as backtracking out of order can also be tried. For example, a speech understanding program may parse a sentence incorrectly, discover its error only at the end

of the sentence, but also discover most of its work was correct. If the operations for each segment have been collected into events, the event for the incorrect segment could be backtracked or undone without destroying the rest of the parse. That segment could then be reworked in view of the new knowledge of the rest of the sentence and results combined with the original work.

Finally, this entire mechanism can be used by user programs to create and manipulate context independent of their program structure. This facility is playing an important role in programs that synthesize plans with conditional statements and theorem provers that use hypothetical deduction methods.

II ALGORITHMS

A. Terminology

A context is represented by a dotted pair (d . b), where d is the dynamic context and b is the backtracking context. d and b are both simple lists of atoms that may be generated in a nonrecurring fashion. Numbers are currently used. For example

(2 1).(3 2 1) is a context

(2 1) is the dynamic context

2 is the dynamic head

(3 2 1) is the backtracking context

3 is the backtracking head .

B. Internal Expression Form

An expression is a property list. Those properties that may change are not stored in the normal property list way. Rather, the dynamic head and the backtracking head are used as property names, forming a path to a standard name-value list form. If we stored property PROP1 under name IND1 on expression e we might find

e = (NETEXPRESSION I1 P1 I2 P2 ... 2 ((3 IND1 PROP1)) 111) .

If we then stored PROP2 under IND2, e would become

(... 2 ((4 IND1 PROP1) (3 IND1 PROP1 IND2 PROP2)) ...) .

Thus the dynamic heads become names on the top level of e, while backtracking heads become the names of associated pair lists.

C. Retrieval

The property-retrieval function is given a name, an expression, and a context, and it is expected to return a property. The search occurs in two phases. The first phase uses the dynamic context to choose a sublist from the top level of the expression. The choice is made by first considering the dynamic head. If it occurs as an indicator, the corresponding sublist is chosen; if not, the next element of the dynamic context is tried. This continues until a sublist is found. If our context were

(3 2 1).(4 3 2 1)

and we used the final e given above, this first phase would find the sublist

((4 IND1 PROP1) (3 IND1 PROP1 IND2 PROP2))

as the property for the name (and dynamic context element) 2. This search process is speeded up by always inserting new dynamic names at the front of expressions and stepping through the dynamic context and the expression in parallel until a common element is found.

The final phase uses the backtracking context in a similar way to find the appropriate list of name-value pairs. It first searches for a list headed with the backtracking head. If that fails, it uses the next element of the backtracking context and continues this way until a sublist is found. In our example this would produce (4 IND1 PROP1).

If no element of the backtracking context is present, the first phase is reentered with the next dynamic element.

After a list is found beginning with a backtracking element, it is searched for the appropriate name. If one is found, the corresponding property is returned. If one is not found, the search for another sublist continues using the next element of the backtracking context. If we were searching for the property under IND2 in our example, the initial sublist search would fail, and the search using the second backtracking context element would produce

(3 IND1 PROP1 IND2 PROP2) .

Here the search for IND2 would succeed and we would find PROP2.

D. <u>Storage</u>

The storage function also works in two phases. Given an expression, name, property, and context, it stores the property. The first phase uses the dynamic context to find a top-level sublist of the expression. The search is identical to the first phase of the retrieval function. The second storage phase, however, does not perform a repetitive search. The associated pair list that results from the first phase is searched for the backtracking head. If a sublist is not found for the backtracking head, one is immediately generated and the name-value pair put on it. Thus, under each new backtracking context, old values are not destroyed, and they may be retrieved later by restoring the context to its previous state.

III EXAMPLES

A. Introduction

The interaction of the dynamic and backtracking contexts can become quite complicated. Basically, contexts are modified during one of the following:

- Function calls and returns
- Creation of cooperating processes
- Creation of parallel processes
- Passing of backtracking points.

The following 11 examples illustrate the interplay of these processes. The examples only discuss the binding of variables to values, but the same procedures are used to handle arbitrary properties of expressions. The figures show the flow of control. Each major control point is labeled in sequence: T1, T2, etc. Other notations are explained in the examples. There are no loops in the programs. Backtracking, however, causes control to be restored at previous (higher) points in the figures. When possible, indentation is used to clarify this procedure.

B. Function Calls

1. Simple Function Calls--Example 1

Figure III-1 shows the role of the context mechanism during common subroutine calls. Assume the global context is (1).(1) when subroutine A is called. The following is a protocol of the rest of the execution.

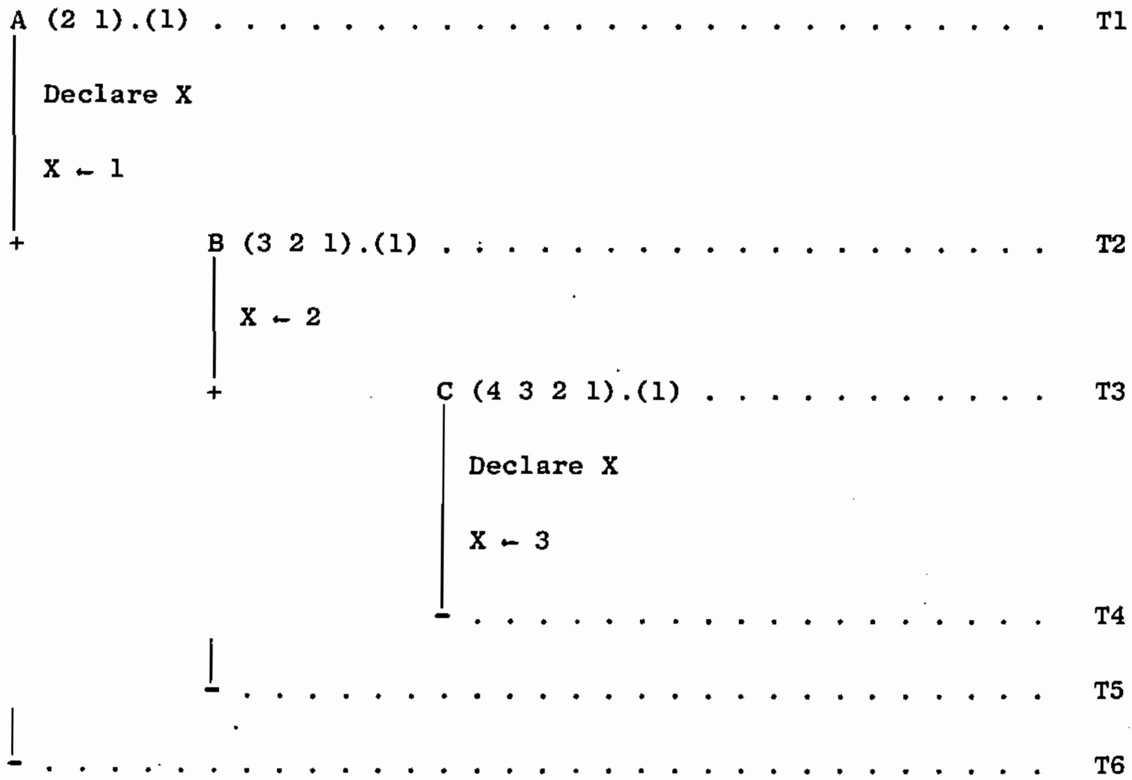


FIGURE III-1 SIMPLE FUNCTIONS CALLS

T1 The dynamic context is increased to (2 1).

X is declared local. Its property list takes the form

(... 2 ((1 NETVALUE NOSUCHPROPERTY)) ...).

X is assigned the value 1. Its property list is changed

to (... 2 ((1 NETVALUE 1)) ...).

T2 Subroutine B is called. The dynamic context is increased

to (3 2 1).

X is assigned the value 2. Notice that X is global to

B. Its property list takes the form

(... 2 ((1 NETVALUE 2)) ...).

T3 Subroutine C is called. The dynamic context is increased

to (4 3 2 1).

X is declared local. Its property list takes the form

(... 4 ((1 NETVALUE NOSUCHPROPERTY)) ((2 NETVALUE 2)) ...).

X is assigned the value 3. Its property list is changed

to (... 4 ((1 NETVALUE 3)) 2 ((1 NETVALUE 2)) ...).

T4 Subroutine C returns. The dynamic context is restored to

(3 2 1).

The indicator 4 and the property ((1 NETVALUE 3)) on the

top level of X are now garbage, and will be collected

later. They are made garbage merely by removing 4 from

the list of currently active dynamic context numbers.

They have become garbage because the dynamic indicator

4 was created solely for this incarnation of subroutine C. The subroutine is finished, and making them garbage is equivalent to popping them off a push-down stack in a more conventional system.

T5 Subroutine B returns. The dynamic context is restored to (2 1).

Notice that since X was global in B, the value was changed under A's context.

T6 Subroutine A returns. The dynamic context is restored to (1). The indicator 2 and the property ((1 NETVALUE 2)) now become garbage.

2. Simple Backtracking--Example 2

Figure III-2 shows the role of the context mechanism during the execution of a simple backtracking program. Assume the global context is (1).(1) when subroutine A is called. The following is a protocol of the rest of the execution.

T1 The dynamic context is increased to (2 1).

X is declared local. Its property list takes the form (... 2 ((1 NETVALUE NOSUCHPROPERTY)) ...).

X is assigned the value 1. Its property list takes the form (... 2 ((1 NETVALUE 1)) ...).

T2 A backtracking point is passed. The backtracking context is increased to (2 1).

```
(1).(1)

A (2 1).(1) . . . . . . . . . . . . . . . . . . . . . . . .   T1

  Declare X

  X ← 1

• (2 1).(2 1) . . . . . . . . . . . . . . . . . . . . . . .   T2

        (2 1).(3 1) . . . . . . . . . . . . . . . . . .   T4

              (2 1).(4 1) . . . . . . . . . . . . . .   T6

  X ← X + 1

FAIL . . . . . . . . . . . . . . . . . . . . . . . .   T3,T5,T7

- . . . . . . . . . . . . . . . . . . . . . . . . .        T8
```

FIGURE III-2 SIMPLE BACKTRACKING

1 is added to X and the result assigned to X. Notice that the value of X is retrieved from backtracking context (2 1). The form of X's property list is now (...2 ((2 NETVALUE 2) (1 NETVALUE 1)) ...).

T3 The program fails.

Control is returned to the last backtracking point.

T4 Assume that a second alternative is available at the backtracking point. The backtracking context is changed to (3 1). Since the backtracking context 2 will never again be used, the change to X under T2 is effectively erased and the list (2 NETVALUE 2) is now garbage.

X is again increased by 1. Just as before, the value is read under backtracking context (1), but this time the value is stored under backtracking context (3 1). The property list of X now takes the form (...2 ((3 NETVALUE 2) (2 NETVALUE 2) (1 NETVALUE 1)) ...).

T5 The program fails again.

Control is returned to the last backtracking point.

T6 Assume that a third alternative is available at the backtracking point. The backtracking context is changed to (4 1).

X is increased for the third time, but still assigned the value 2.

T7 This time the program does not fail and control proceeds.

T8 The subroutine returns.

The dynamic context is restored to (1).

Everything under property 2 on X's property list now becomes garbage.

3. Multiple Backtracking--Example 3

Figure III-3 shows what happens when there are no alternatives left at a backtracking point. The program begins just like example 2.

T3 The second point is passed. The backtracking context is increased to (3 2 1).

T4 Failure occurs and control is returned to the last backtracking point.

T5 A second alternative is chosen and the backtracking context is updated.

T6 Failure occurs for the second time. Control is again returned to the last backtracking point.

T7 No alternatives are available, so failure occurs and control is returned to the first backtracking point.

T8 The second alternative is chosen at the first point, and the backtracking context is updated.

T9 The second backtracking point is passed for the second time. An alternative is chosen and the backtracking context is increased.

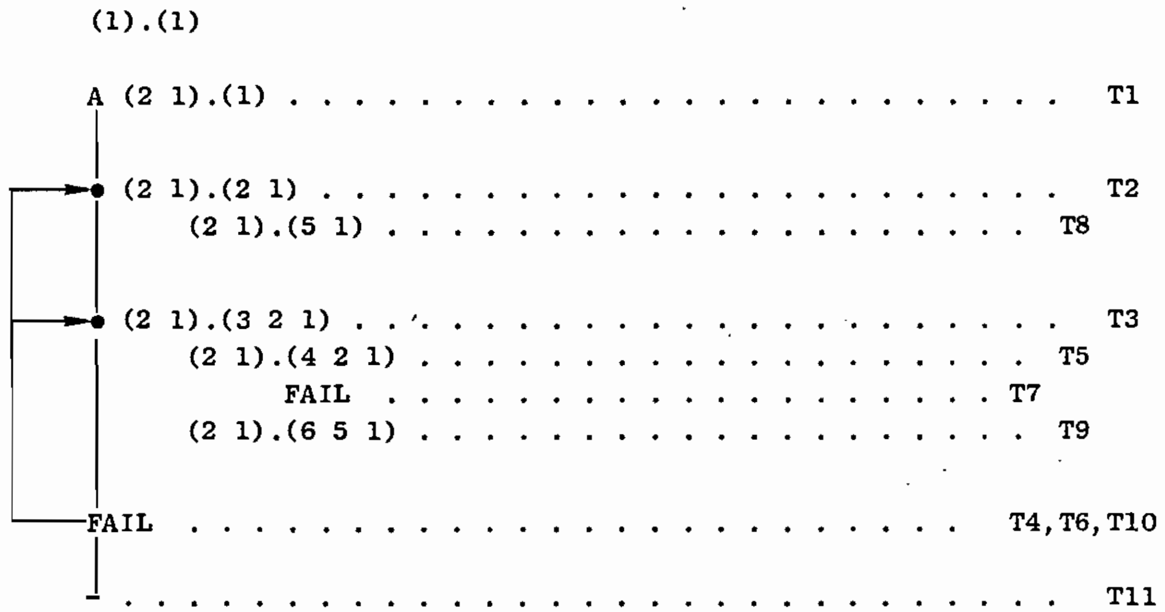


FIGURE III-3 MULTIPLE BACKTRACKING

T10 This last time, no failure occurs and the program proceeds.

T11 The subroutine succeeds and returns to the calling program.

4. Function Calls with Backtracking--Example 4

Figure III-4 shows the interaction of the dynamic and backtracking contexts. The program begins the same way as examples 2 and 3.

T3 Subroutine B is entered. The dynamic context is increased.

T4 Subroutine C is entered. The dynamic context is increased again.

T5 A backtracking point is passed. An alternative is chosen, and the backtracking context is increased to (3 2 1).

T6 Subroutine C has returned, and a backtracking point is passed in subroutine B. An alternative is chosen and the backtracking context is increased to (4 3 2 1). Notice that the backtracking context was not reduced during the return from the subroutine while the dynamic context was reduced.

T7 Subroutine B has returned to subroutine A, and a failure occurs. Control is returned to the last backtracking point, the middle of subroutine B in the state when the point was originally passed. This state was preserved by the context mechanism when new backtracking contexts were created.

T8 The second alternative is chosen, and the backtracking context is updated.

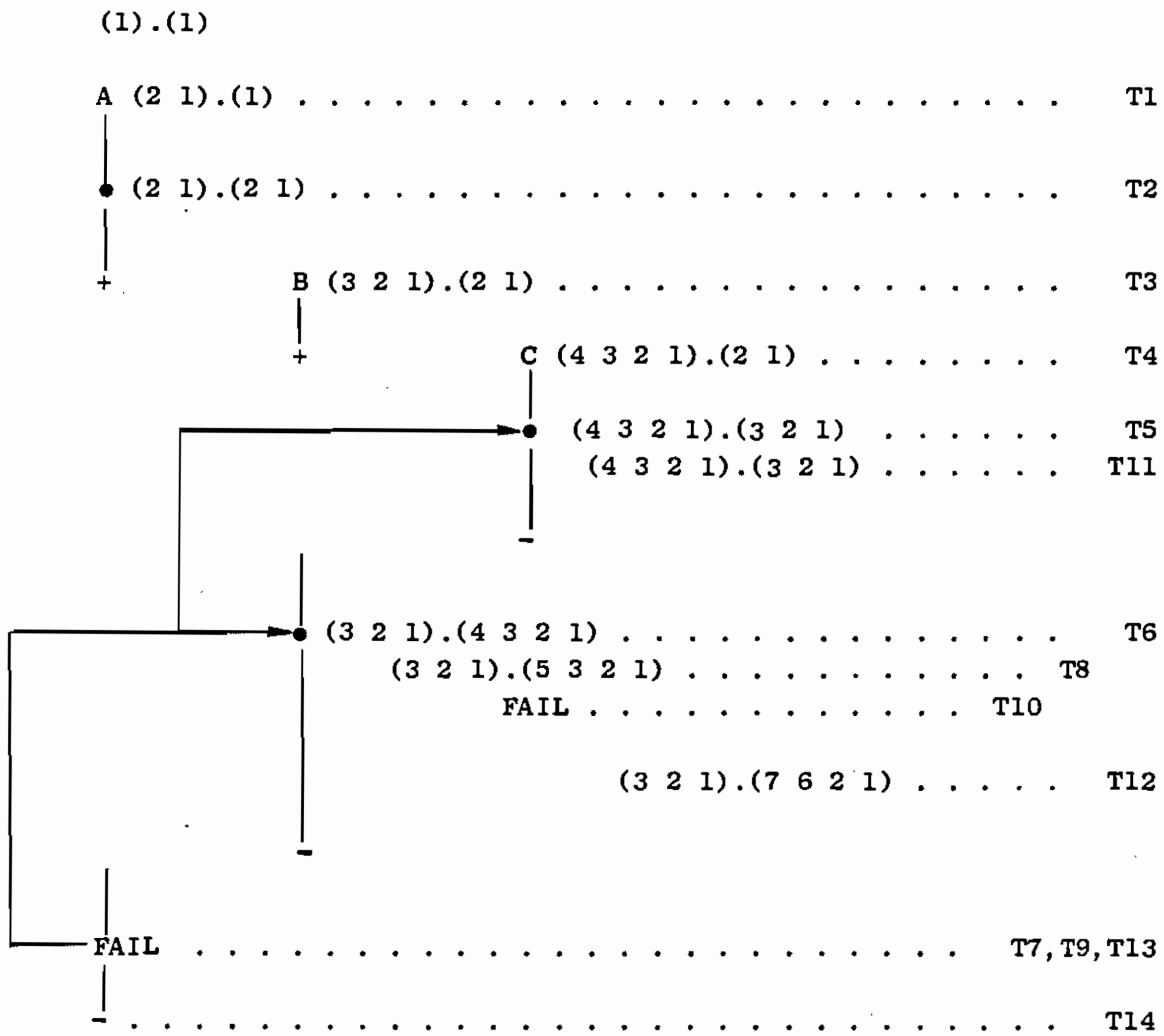


FIGURE III-4 FUNCTION CALLS WITH BACKTRACKING

T9 Subroutine B has returned to subroutine A for the second time. Another failure occurs, and control is again returned to subroutine B. B is restored to the same state is was in the first time the backtracking point was passed.

T10 No alternatives are available and the backtracking point fails. Control is now restored to subroutine C.

T11 The second alternative is chosen, and the backtracking context is updated.

T12 This second backtracking point is now passed for the second time. The first alternative is chosen, and the backtracking context increased.

T13 No failure occurs at this time, control flows on.

T14 Subroutine A succeeds and thus returns.

C. Cooperating Processes

Examples 5 through 9 deal with cooperating processes. The characteristic that differentiates these programs from standard functions is that they transfer control back and forth, always retaining their state. "Coroutine" as well as other names have been applied to these kinds of programs. These names have attempted to distinguish parallel cooperation, subordinate cooperation, and other structures. We make no such formal distinction and call them all cooperating processes.

1. Simple Cooperating Processes--Example 5

Figure III-5 shows how the dynamic context is used to create and use a cooperating process. As in the first example, the global context is (1).(1).

T1 Subroutine A is called. The dynamic context is increased to (2 1).

T2 Process B is created. A new dynamic context, (3 2 1), is created and assigned to it. The process is not yet run.

T3 Process B is called. Any local variables are modified under the dynamic context (3 2 1). The current backtracking context becomes B's backtracking context.

T4 B returns to A. The dynamic context is restored to (2 1). Unlike function calls, B's dynamic context is not discarded and garbage is not generated. Everything is saved for further processing with B. Notice that A cannot reference B's local variables without special context processing.

T5 A again calls B. The dynamic context (3 2 1) is again restored. Nothing within B has been lost.

T6 B returns to A. A's dynamic context is restored.

T7 A returns to the calling program.

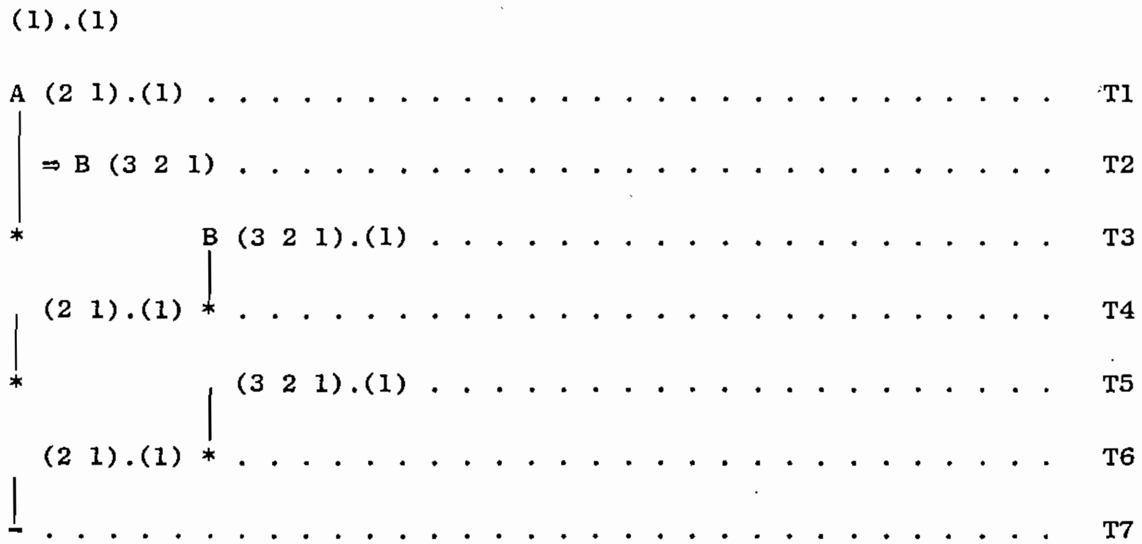


FIGURE III-5 SIMPLE COOPERATING PROCESS

2. Coroutines--Example 6

Figure III-6 shows the creation and running of ordinary coroutines.

T1 Subroutine A is called and started in execution.

T2 Coroutine B is created.

T3 Coroutine C is created.

T4 B is entered.

T5 C is entered from B.

T6 C returns to B.

T7 B returns to C.

T8 C returns to B. All these returns have been RESUMEs, they are not returns.

T9 B does a normal return to A.

3. Cooperating Processes with Backtracking (1)--Example 7

Figure III-7 shows the running of a simple subordinate process with backtracking to the superordinate process.

T1 A is called.

T2 B is created.

T3 B is entered. It is assigned A's backtracking context.

T4 B returns to A.

T5 A passes a backtracking point.

T6 A returns to B. B is again assigned A's current backtracking context.

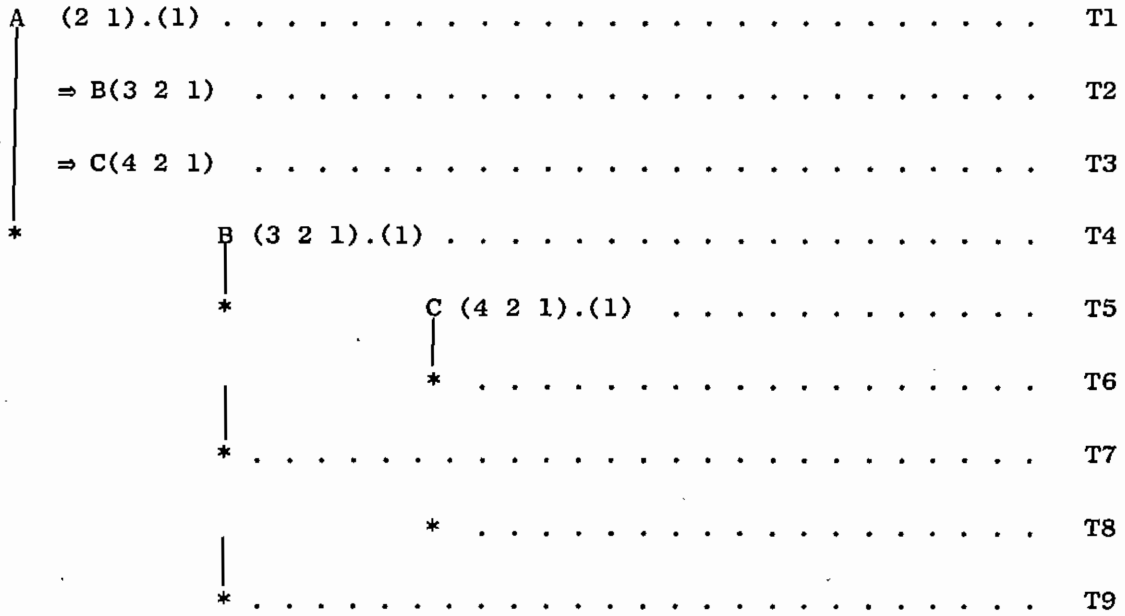


FIGURE III-6 COROUTINES

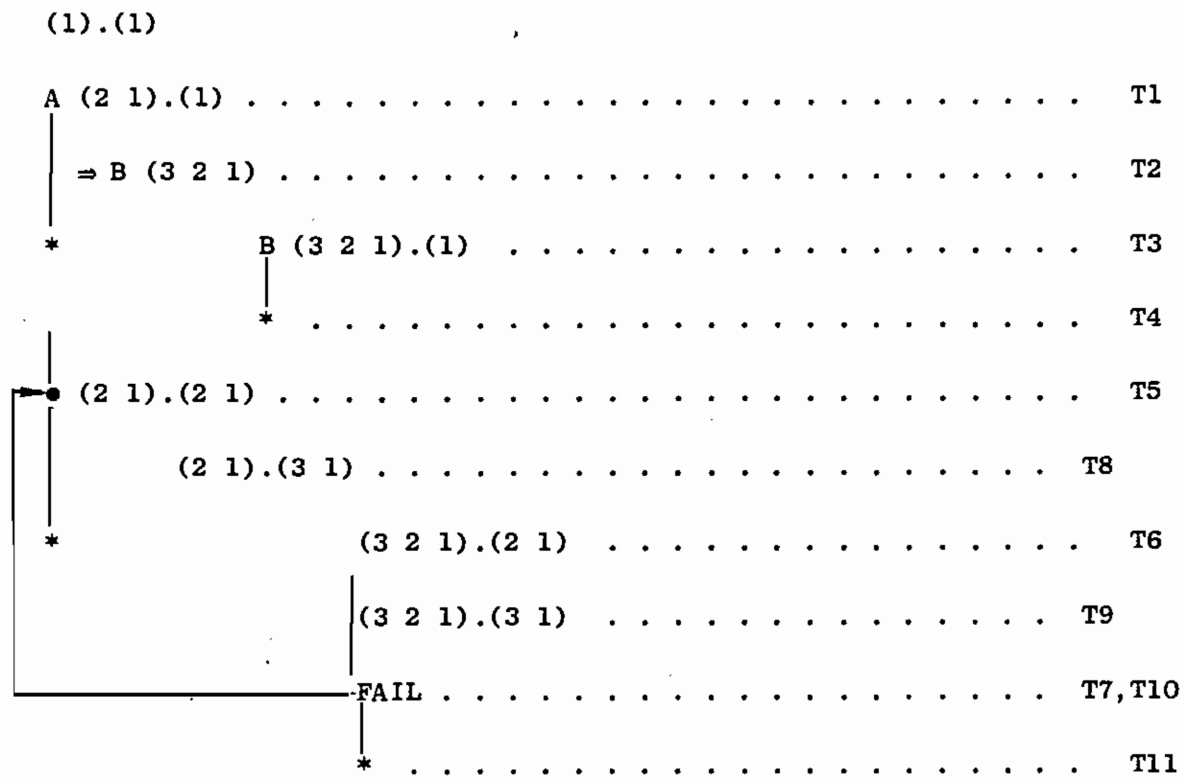


FIGURE III-7 COOPERATING PROCESSES WITH BACKTRACKING (1)

T7 B fails. Control is restored to the last backtracking point.

T8 The backtracking context is modified and execution continues.

T9 A again returns to B. B is assigned A's new current backtracking context.

T10 B does not fail and control continues.

T11 B returns to A.

4. Cooperating Processes with Backtracking(2)--Example 8

Figure III-8 shows, like Figure III-7, the running of a simple subordinate process with backtracking to the superordinate process. However, the running concludes with the subordinate process passing a backtracking point (to illustrate passing on of backtracking context).

T1 A is called.

T2 B is created.

T3 A passes a backtracking point.

T4 B is entered.

T5 B returns to A.

T6 A returns to B.

T7 B fails.

T8 The backtracking context is modified.

T9 B is again entered.

T10 B again returns to A.

T11 A again returns to B.

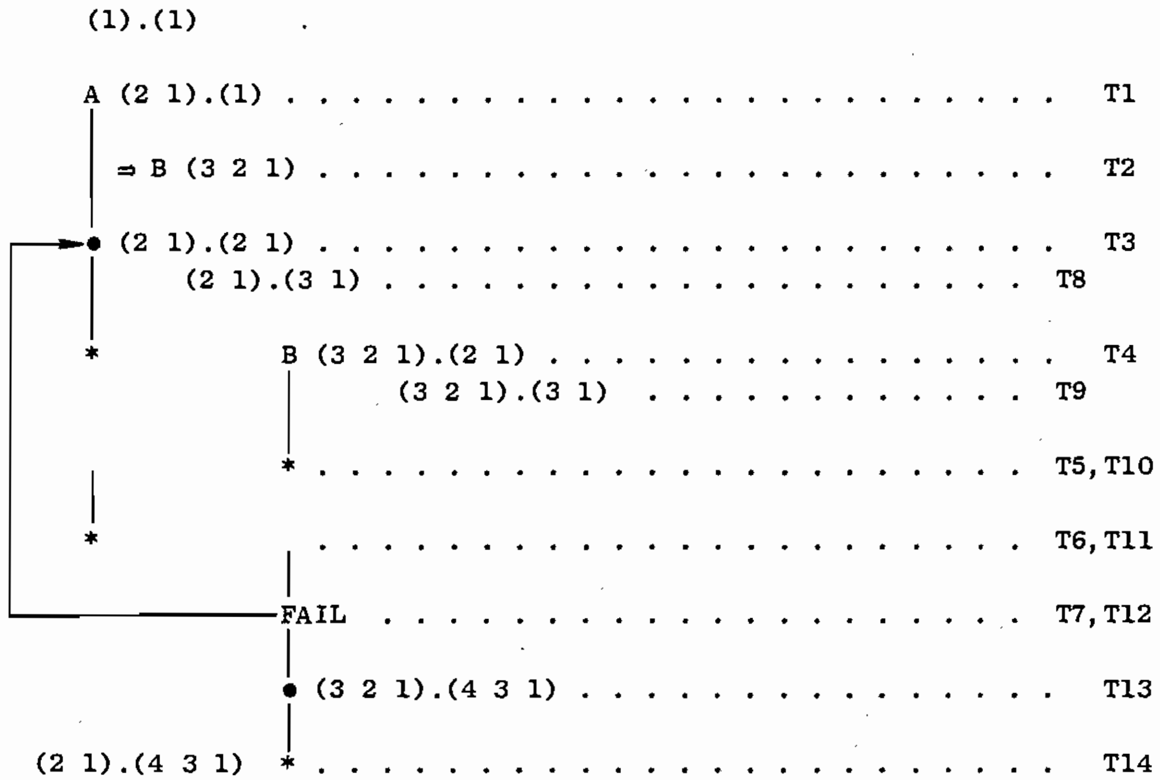


FIGURE III-8 COOPERATING PROCESSES WITH BACKTRACKING (2)

T12 B does not fail. Control continues.

T13 B passes a backtracking point.

T14 B returns to A. A's backtracking context is set to B's, while A's dynamic context is set to its original state.

5. Cooperating Processes with Backtracking (3)--Example 9

Figure III-9 shows the creation of two subordinate processes. The backtracking occurs between these two programs, rather than back to the superordinate process as in examples 7 and 8.

T1 A is called.

T2 B is created.

T3 C is created.

T4 B is entered.

T5 B passes a backtracking point.

T6 B returns to A. Notice the transfer of the backtracking context.

T7 C is entered from A. C is assigned A's backtracking context.

T8 C fails. Control is transferred back to last backtracking point in B.

T9 The backtracking context is modified.

T10 B again returns to A. A is assigned the new backtracking context.

T11 A again returns to C. C is assigned the new backtracking context.

T12 C does not fail and control continues.

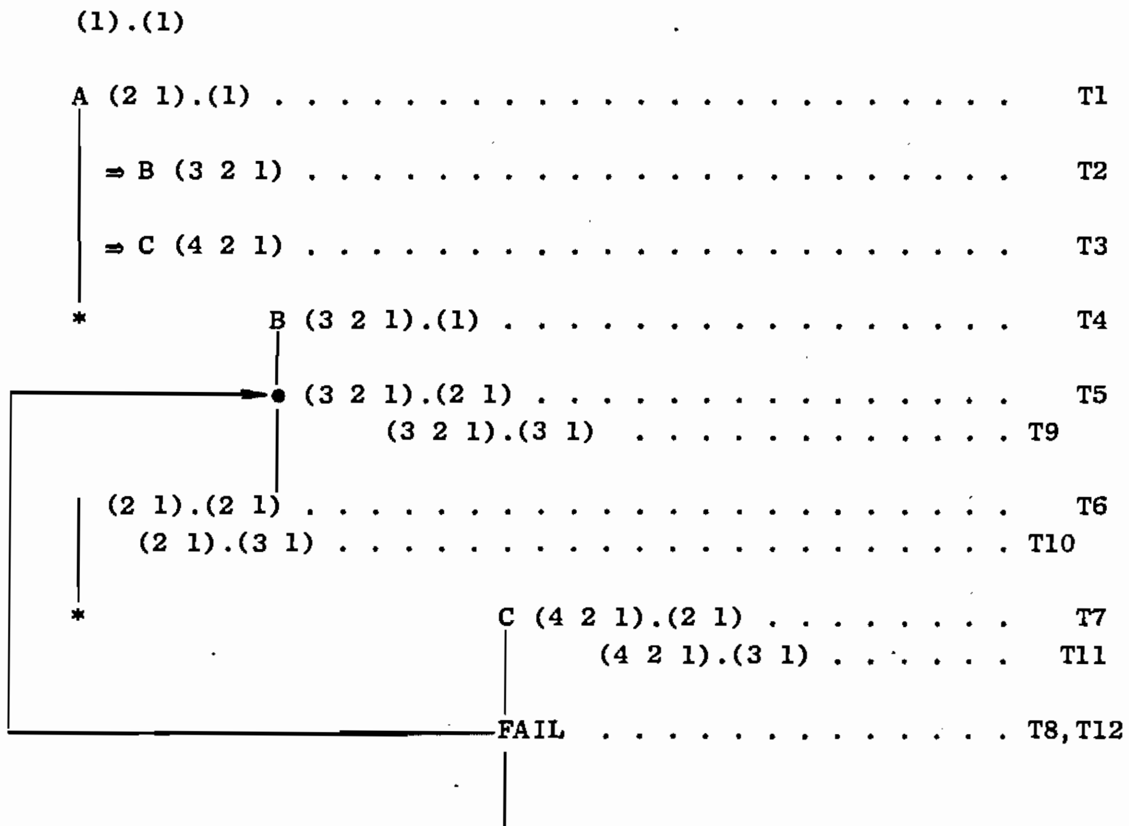


FIGURE III-9 COOPERATING PROCESSES WITH BACKTRACKING (3)

D. Parallel Processes

Examples 10 and 11 deal with parallel processes. As will be seen, the current definition is strict, and does not appear to be useful. More experimentation with problem-solving systems will help bring out the needs of cooperating parallel processes.

1. Simple Parallel Processes-Example 10

Figure III-10 shows the creation and running of two parallel programs. It also illustrates that the parent process can be run in parallel with the two subordinate processes.

T1 A is called.

T2 B is created.

T3 C is created.

T4 A, B, and C are all started in parallel. A new backtracking context is assigned to each one. This means that A's state at this point becomes and remains the global state for B and C even though A, B, or C might change a common global variable. Notice that the procedures cannot affect each other in any way.

T5 At some time, maybe even all together, all three processes change the global variable X. Each changes it in his own way. Thus, the programs are truly parallel, but cannot communicate with each other.

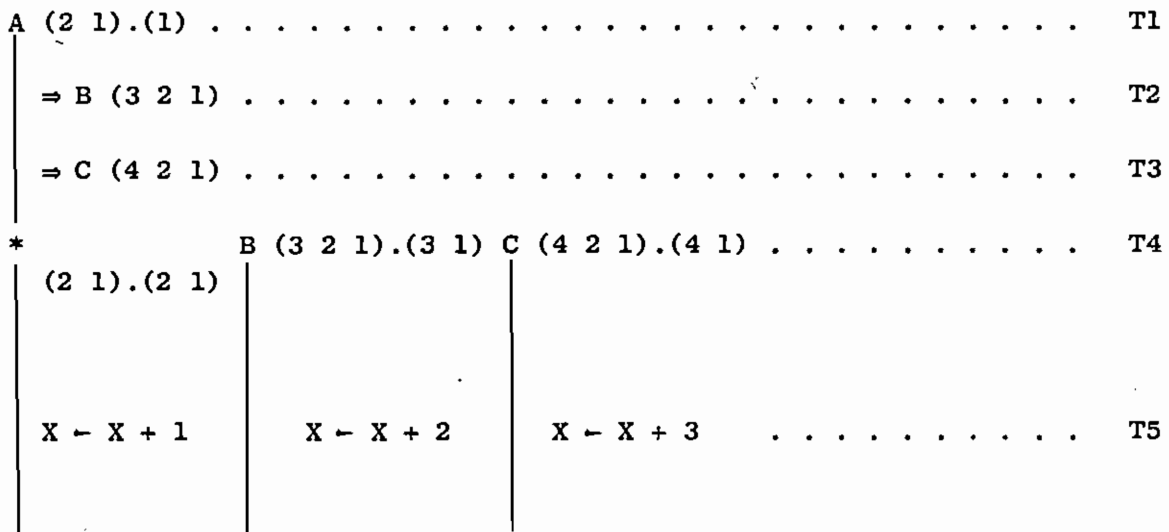


FIGURE III-10 SIMPLE PARALLEL PROCESSES

2. Parallel Processes with Backtracking--Example 11

Figure III-11 shows the creation of a subordinate parallel process with failure back to the superordinate process.

T1 A is called.

T2 B is created.

T3 A passes a backtracking point.

T4 B is started in parallel with A.

T5 B fails.

T6 Control is restored to the backtracking point. The backtracking context is modified.

T7 B is again started in parallel with A. Notice that each is again assigned a new backtracking context.

T8 B does not fail and both programs proceed.

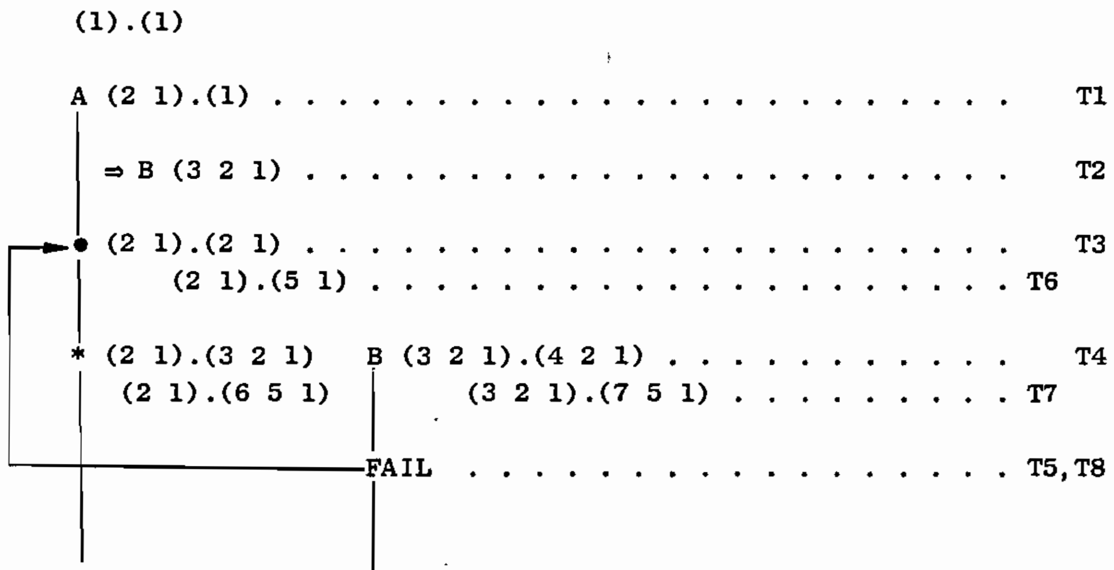


FIGURE III-11 PARALLEL PROCESSES WITH BACKTRACKING

IV IMPLEMENTATION

There are two main parts of the context mechanism: the garbage collector and the store and fetch routines. The garbage collector is of no concern to the user since its operations are self-contained and cannot be modified. The names of general store and fetch routines all start with the prefix CTX:. There are only a few routines, and each is small and simple. In the following description of the programs, the letter used for the argument indicates the type of the argument. The correspondence is

E	any general QA4 expression
I	an indicator name--should be a LISP atom
P	a property, any LISP expression
C	a context.

The functions with CTX: as prefix are:

PUT(E,P,I,C)	Puts P on E under I and C.
GET(E,I,C)	Gets a P from E using I and C.
XD(C)	Increases the dynamic context of C and returns the new C.
POPD(C)	Removes the dynamic head of C and returns the new C.
POPB(C)	Removes the backtracking head of C and returns the new C.
SETUP	Initializes the context mechanism.
DYN(E,C)	Searches the top-level of E for the appropriate sublist defined by the dynamic context of C. Used by both GET and PUT.

DECLARE(E,C) Sets up the expression E so that it is local in the context C.

The global variables DCTXNDX, BCTXNDX, VDCTX, and BVCTX are used to generate new context numbers and keep track of all currently valid context numbers for the garbage collector. These names come from: D for dynamic, B for backtracking, CTX for context, NDX for index, and V for valid.

V SUMMARY

A. Space and Garbage Collection

There are two fundamental drawbacks with the context mechanism: (1) the size of storage and (2) garbage collection time. The amount of storage needed to store a value for a variable is about three times that required in a stack system. This should be expected, because the state is dispersed and thus each value must have state information associated with it. Garbage collection time is also greatly increased. This is due to the fact that, with a concentrated state system (Floyd 1967), the variable bindings for entire states may be freed with a single stack operation. With a context mechanism, however, the state information for each variable binding must be examined separately. Both of these inefficiencies appear to be inherent in the context approach, and there is no known way of overcoming them.

B. Binding Retrieval Time

The QA4 implementation also extends the store and fetch time by a considerable but unmeasured amount. This inefficiency, however, could be overcome by more extensions to the QA4 system.

For interpreted code, a search must take place for variable bindings in both types of systems. In LISP, for example, the stack must be searched, while in QA4 the property list of the variable must be searched. Both these searches, moreover, scan block structured entities. Thus scan techniques, such as zeta coding, that improve one can be used for the other.

References for nonlocal variables in compiled code must be made through similar scan techniques for both approaches. References to local variables in compiled code, however, are made directly to a stack position in a stack-oriented system. It does not appear that this efficiency can be matched with a context mechanism, but it can be closely approximated. Each variable could have an indirect cell associated with it that could store a pointer to the last referenced value for the variable. Then each reference would first compare the context code in the indirect cell with the context code for this reference and use the indirect cell with the context code for this reference and use the indirect pointer if it applied. If it did not apply, a search could be made and the indirect cell updated after the search. This would only cause a loss of time for the first reference to a local variable. The compare operations could be a part of the hardware, just as stack operations have been incorporated into hardware. With a machine of this sort, the binding retrieval time could be equivalent to that of stack systems.

C. Versatility

The major advantage of the context mechanism is its implementation simplicity and the ease of experimenting with the interpreter. The QA4 language is primary a research tool to help evolve a semantics for a language for writing problem-solving systems. Thus, experimentation and change are vital for success. This context mechanism has greatly aided this developmental type research. Various styles of backtracking can be

tried in conjunction with different process interactions. It is especially difficult with stack systems to backtrack over RESUME statements between independent process structures, and such systems dictate a single interpretation for backtracking. A most important kind of experimenting has occurred because a unified system was used for both the interpreter and the user WRT options. This has led to the current WRT semantics and will, we expect, lead to even more useful semantics in the future.

Future areas of experimentation will include not only the semantics of dynamic process structures, but static data structures. That is, data structures that represent areas of knowledge where words or expressions have meanings that vary between the areas. The problem we will study will be the interaction of these dynamic and static structures. The context mechanism appears to provide the versatility for such a study.

REFERENCES

REFERENCES

D. G. Bobrow and B. Wegbreit, "A Model and Stack Implementation of Multiple Environments," Bolt, Beranek and Newman, Cambridge, Massachusetts (1972).

M. E. Conway, "Design of a Separable Transition-Diagram Compiler," C. ACM, Vol. 6, pp. 396-408 (1963).

R. W. Elliot, "A Model for a Fact Retrieval System," Ph.D. Thesis, The University of Texas, Austin, Texas (1965).

B. Elspas, M. W. Green, K. N. Levitt, and R. J. Waldinger, "Research In Interactive Program-Proving Techniques," SRI Project 8398, Phase II, Stanford Research Institute, Menlo Park, California (May 1972).

R. E. Fikes and N. J. Nilsson, "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving," Artificial Intelligence, Vol. 2, Nos. 3/4, pp. 189-208 (1971).

R. W. Floyd, "Nondeterministic Algorithms," J. ACM, Vol. 14, pp. 636-644 (October 1967).

C. C. Green, "Applications of Theorem Proving to Problem Solving," Proc. International Joint Conference on Artificial Intelligence, D. E. Walker and L. M. Norton, Eds., Washington, D.C., pp. 219-239 (May 1969).

L. Golumb and L. Baumert, "Backtrack Programming," J. ACM, Vol. 12, No. 4, pp. 516-524 (1965).

C. Hewitt, "Description and Theoretical Analysis (Using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot," Ph.D. Thesis, Department of Mathematics, Massachusetts Institute of Technology, Cambridge, Massachusetts (1972)

D. Kalish and R. Montague, Logic, Techniques of Formal Reasoning, Harcourt, Brace, and World, Inc., Chapter 1, p. 16 (1964).

R. E. Kling, "A Paradigm for Reasoning by Analogy," Artificial Intelligence, Vol. 2, No. 2, pp. 147-178 (1971).

P. J. Landin, "The Mechanical Evaluation of Expressions," Computer Journal, Vol. 6, pp. 308-320 (1963/64).

J. McCarthy, "Programs with Common Sense," in "Mechanization of Thought Processes," Proc. Symposium National Physical Laboratory, Vol. 1, pp. 77-84 (1958). Also, Stanford Artificial Intelligence Project Memo No. 7, Stanford University, Stanford, California.

J. McCarthy, "A Basis for a Mathematical Theory of Computation," Computer Programming and Formal Systems, Braffort and Hirschberg, Eds., North-Holland, pp. 33-70 (1963).

J. McCarthy, et al., LISP 1.5 Programmer's Manual, M.I.T. Press, Cambridge, Massachusetts (1962).

J. McCarthy and P. Hayes, "Some Philosophical Problems from the Standpoint of Artificial Intelligence," in Machine Intelligence 4, B. Meltzer and D. Michie, Eds., American Elsevier Publishing Co., Inc., pp. 463-502 (1969).

V. McDermott and G. J. Sussman, "The Conniver Reference Manual," Massachusetts Institute of Technology Artificial Intelligence Laboratory, Memo No. 259, Cambridge, Massachusetts (May 1972).

Z. Manna and R. J. Waldinger, "Toward Automatic Program Synthesis," C. ACM, Vol. 14, No. 3, pp. 151-165 (March 1971).

M. Minsky, "Steps Toward Artificial Intelligence," in Computers and Thought, E. Feigenbaum and J. Feldman, Eds., McGraw-Hill Book Co., Appendix A, p. 5: Property Lists (1963).

J. Piaget, Genetic Epistemology, Columbia University Press, New York, p. 21 (1960).

W. V. O. Quine, Mathematical Logic, Harvard University Press, Cambridge, Massachusetts, pp. 33-37 (1965).

B. Raphael, "A Computer Program which 'Understands'," AFIPS Conf. Proc., Vol. 26, Pt. 1, pp. 577-587 (1964).

J. A. Robinson, "Mechanizing Higher-Order Logic," in Machine Intelligence 4, B. Meltzer and D. Michie, Eds., American Elsevier Publishing Co., Inc., pp. 151-170 (1969).

T. A. Winograd, "Procedures as a Representation for Data in a Computer Program for Understanding Natural Language," Ph.D. Thesis, Department of Mathematics, Massachusetts Institute of Technology, Cambridge, Massachusetts (1971).

W. Teitelman, D. G. Bobrow, A. K. Hartley, and D. L. Murphy, "BBN-LISP TENEX Reference Manual," Bolt, Beranek and Newman, Inc., Cambridge, Massachusetts (1972).

W. Teitelman, "Design and Implementation of FLIP, a LISP Format Directed List Processor," Bolt, Beranek and Newman, Inc., Cambridge, Massachusetts (1967).