N73 25173

Interim Scientific Report
7 October 1969 to 15 April 1970

RESEARCH AND APPLICATIONS - ARTIFICIAL INTELLIGENCE

By: L. J. CHAITIN R. O. DUDA P. A. JOHANSON
B. RAPHAEL C. A. ROSEN R. A. YATES

Prepared for:

NATIONAL AERONAUTICS AND SPACE ADMINISTRATION
600 INDEPENDENCE AVENUE, S.W.
WASHINGTON, D.C. 20546
Attention: MR. SAMUEL A. ROSENFELD/RET

CONTRACT NAS12-2221

CASE FILE
COPY

Sponsored by

ADVANCED RESEARCH PROJECTS AGENCY
WASHINGTON, D.C. 20301
ARPA ORDER 1058 AMENDMENT 1

STANFORD RESEARCH INSTITUTE
Menlo Park, California 94025 · U.S.A.

STANFORD RESEARCH INSTITUTE
Menlo Park, California 94025 · U.S.A.

Interim Scientific Report
7 October 1969 to 15 April 1970

April 1970

RESEARCH AND APPLICATIONS - ARTIFICIAL INTELLIGENCE

By: L. J. CHAITIN R. O. DUDA P. A. JOHANSON
B. RAPHAEL C. A. ROSEN R. A. YATES

Prepared for:

NATIONAL AERONAUTICS AND SAPCE ADMINISTRATION
600 INDEPENDENCE AVENUE, S.W.
WASHINGTON, D.C. 20546
Attention: MR. SAMUEL A. ROSENFELD/RET

CONTRACT NAS12-2221

SRI Project 8259

Approved by:

DAVID R. BROWN, *Director*
Information Science Laboratory

BONNAR COX, *Executive Director*
Information Science and Engineering Division

Sponsored by

ADVANCED RESEARCH PROJECTS AGENCY
WASHINGTON, D.C. 20301
ARPA ORDER 1058 AMENDMENT 1

Copy No. ...90...

ABSTRACT

The primary objectives of this program are to investigate and develop techniques in artificial intelligence and apply them to the control of a mobile automaton, enabling it to carry out tasks, autonomously, in a realistic laboratory environment.

As part of technique developments, there are reports of progress in scene analysis, and short-term and long-term problem solving.

Scene analysis is aimed at providing the automaton system with its primary source of information about its environment. Such information (primarily visual) must be entered into appropriate internal models in a form useful for the problem-solving and planning systems. Several parallel approaches--line analysis and region analysis methods--are discussed, together with how knowledge of the actual environment is used to help interpret the processed information.

The short-term problem-solving research is primarily aimed at providing new tools and concepts for coping with tasks of increased complexity to be implemented in the next year. A major software tool, QA3 (a question-answering system using theorem proving by resolution), will be revised and upgraded, permitting experimentation with new strategies and also allowing the theorem prover to be used in planning. Underway are implementations of a new n-tuple model and set of operators--routines that, when executed according to some plan, cause the robot to perform specific actions. Both primitive and more complex operators are being defined; these are callable in correct sequence by a planner. The planner organization is the subject of considerable research, which includes

the possible use of a GPS-like deductive mechanism, perhaps together with the revised QA3 system. Finally, executive routines are being developed to supervise execution of operator sequences and updating of models, assessing cost-effectiveness of plans, making decisions such as regarding the need for new sensory information or abandonment of further planning in favor of execution, etc.

The long-term problem-solving research is devoted to design and implementation of a general-purpose formal problem-solving system, QA4, based on mechanized theorem proving in higher-order logic. It is being designed to emphasize the role of semantic information processing and flexible control strategies. It will provide a rich language permitting syntax and semantics of any part of the system to be expressed in its own language, and will include set operations bridging logical and computational operations. It is expected to provide tools and techniques suitable for long-range research in such diverse fields as automatic program writing and robot planning.

CONTENTS

ABSTRACT iii

LIST OF ILLUSTRATIONS vii

I INTRODUCTION 1

II THE VISUAL-PROCESSING SYSTEM 3

A. Introduction 3
B. Line Analysis 4
C. Region Analysis 7

III THE PDP-10 COMPUTING SYSTEM 13

A. Status of PDP-10/PDP-15 Computer Hardware System . . . 13
B. System Programming 13

1. Monitor Modifications 15
2. PDP-15 Software 16
3. PDP-10 User Programs 16

IV SHORT-TERM PROBLEM SOLVING 19

A. Software Tools 19

1. QA3 19
2. The LISP System 24
3. Interlanguage Communication 25
4. Source Language Flexibility 25

B. Designing a Problem-Solving System 26

1. Problem Definition 27
2. The Model 27
3. Operators 29
4. The Planner 37
5. The Executive 42

CONTENTS (Concluded)

V LONG-TERM PROBLEM SOLVING 47

A. Introduction 47

B. The Logic Language 48

1. The Class of Expressions 48

2. Primitive Operations 52

C. Current Implementation 54

VI HARDWARE AND MAINTENANCE 59

REFERENCES 61

APPENDIX A "PDP-15 Simulator," AI Group Technical Note 25, Stanford Research Institute, Menlo Park, California (April 1970) 63

APPENDIX B "A LISP-FORTRAN-MACRO Interface for the PDP-10 Computer," AI Group Technical Note 16, Stanford Research Institute, Menlo Park, California (November 1969) 77

APPENDIX C "Some Remarks on Resolution Strategies," AI Group Technical Note 28, Stanford Research Institute, Menlo Park, California (April 1970) 91

APPENDIX D "LISP TRACE Package for PDP-10," AI Group Technical Note 27, Stanford Research Institute, Menlo Park, California (April 1970) 101

APPENDIX E "A LISP Implementation of BIP," AI Group Technical Note 22, Stanford Research Institute, Menlo Park, California (February 1970) 109

APPENDIX F "A Cost-Effectiveness Basis for Robot Problem-Solving and Execution," AI Group Technical Note (not yet released), Stanford Research Institute, Menlo Park, California (January 1970) 125

ILLUSTRATIONS

Figure 1 Demonstration of Line-Analysis Approach 8

Figure 2 Elementary Regions in a Digitized Picture 9

Figure 3 PDP-10 Configuration 14

I INTRODUCTION

The primary objectives of this program are, firstly, to investigate and develop techniques in artificial intelligence, and, secondly, to apply them to the control of a mobile automaton carrying out tasks in a realistic environment. These tasks would be such that normally require human intelligence in sensing, problem solving, planning, and execution. By developing artificial intelligence techniques sufficiently general to have wide applicability for Government and industrial use, we shall be able to devise integrated systems capable of replacing humans in situations that are either environmentally hostile or too remote for satisfactory communication and control, or that require very rapid and tireless response to sensed signals.

This project began in October 1969 as a direct continuation of work performed and reported on under previous contracts.[1,2*] This report describes interim results of continuing research in visual scene analysis, short-term problem solving, and long-term problem solving; it also documents the status of the changeover from an SDS-940 to a PDP-10 computer system. A number of additional topics that have been deemed relevant and important are included as appendices.

---

* References are listed at the end of this report.

Page Intentionally Left Blank

II THE VISUAL-PROCESSING SYSTEM

A. Introduction

The vision system for the automaton provides it with a primary source of information about its environment. This information must be obtained from a digitized television picture and from general knowledge of the characteristics of the environment. The information obtained must be entered in the model in a form useful for the problem-solving system.

Our current work is a continuation of two basic approaches followed in the past--line analysis and region analysis. Line analysis exploits the fact that the walls, doorways, and most of the other objects in the automaton's environment have straight-line boundaries. Although noise and limited gray-scale resolution prevent detection of all of these boundaries, it is often possible to isolate, locate, and identify objects from this kind of information. Region analysis exploits the fact that regions of uniform intensity are either significant in themselves, or can be merged together to form significant entities. We have developed various procedures for merging and describing regions, and for using these descriptions to analyze the scene.*

---

* A description of our previous work on the visual system is given in Ref. 2. More detailed technical descriptions of our line-analysis techniques are given in Ref. 3, and our region-analysis techniques are described in Ref. 4 and 5.

Much of our recent work on the vision system has involved converting routines from the SDS-940 to the PDP-10. However, during this conversion process we have modified many of these programs, both for increased generality and increased efficiency. The following sections describe these changes and their consequences for our future work.

B. Line Analysis

The future tasks being planned for the automaton require it to function in the corridors and adjoining rooms. Straight-line boundaries occur throughout this environment, but with fewer constraints than are present in a single room. Furthermore, poorer lighting conditions and the possibility of encountering visually complex office interiors complicate the scene-analysis problems.

One of the most useful pieces of infomation that can be extracted from the corridor picture is the location of floor/wall boundaries. A baseboard tracking routine[2] developed for use in our experimental room has been modified for corridor applications. The same basic steps (acquisition, tracking, and constrained line fitting) are still employed, but some of these procedures have been significantly changed.

The acquisition and tracking routines required only minor changes. As before, the picture is systematically searched column by column from left to right. The intensity values in each column are examined to find any local intensity minima below the horizon line. Each minimum found is given a score determined by its darkness and deviation from ideal width. If the score is sufficiently high, the program switches to the tracking mode, which limits the column search to a neighborhood of the previously acquired point.

When this program was run on some 66 of our corridor pictures, it was found that reasonably good results could be obtained with some minor

readjustments of thresholds. However, the best results were not as good as those obtained regularly in the experimental room. This was due to several factors, including the reduced contrast in the corridors, the greater frequency of interruptions by doorways, increased problems with shadows and reflections, and occasional catastrophic errors arising when the tracker entered a cluttered area such as the interior of an office seen through an open door. Despite these problems, it was decided to leave the tracking program unchanged and to modify the line-fitting program to cope with the new environment.

The basic knowledge of the environment used by the line-fitting routine is that walls are either perpendicular or parallel, and that corridors have a certain minimum width. In terms of the picture, the first requirement means that when floor/wall boundaries are extended to the horizon, they should pass through one or the other of two vanishing points. The x-coordinates, x and x^*, of these vanishing points are related by

$$xx^* = -\frac{f^2}{\cos^2 \varphi}$$

where f is the distance from the lens center to the effective image plane and φ is the tilt angle of the camera.[2] The second requirement is used to screen out reflected baseboards that might be mistakenly identified as close, parallel walls.

The line-fitting routine effectively operates on a binary picture formed from the baseboard coordinates returned by the tracker. This "picture" has at most one point per column, and typically contains a number of fragments of the true baseboard, plus a number of short spurious segments. The length of a segment is important evidence favoring its validity as baseboard, but long pieces of baseboard are frequently

broken into short fragments. Thus, one of the first steps is to try to construct longer segments, called pseudo segments, from the fragments.

This process begins by ordering the segments by length. A straight line is fit locally to the longest segment and extended across the picture. By perturbing the end points of this long line, 25 candidate lines are obtained, and the one that fits the "picture" data best is selected. Typically, this line fits both the original segment and several other segments well, and all such segments are said to be "explained" by the long line. A score that measures the quality of fit is computed for each segment, and all explained segments are flagged. These segments are projected orthogonally onto the long line, and a pseudo segment is defined by the limits of these projections. This process is then repeated for the remaining segments until no segments are unexplained.

At this point the longest pseudo segment is assumed to run along a valid baseboard. This segment is extended to the horizon and is used to define the two vanishing points. The next step is to test the remaining pseudo segments in turn to see if they are either parallel to or perpendicular to the first one. This is done by comparing the sum of the scores for the segments explained by the pseudo segment to the sum obtained under the constraint that the pseudo segment must pass through one or the other of the two vanishing points. If the constrained result exceeds half of the unconstrained result, the pseudo segment is saved.

Various other tests are applied to the pseudo segments, including tests on the minimum allowed length, the maximum allowed number of pseudo segments passing through a given vanishing point, and the minimum allowed three-space distance between pseudo segments representing parallel walls. In addition, we intend to introduce tests to determine whether or not an area is sufficiently cluttered to reject the report of the tracker as being unreliable.

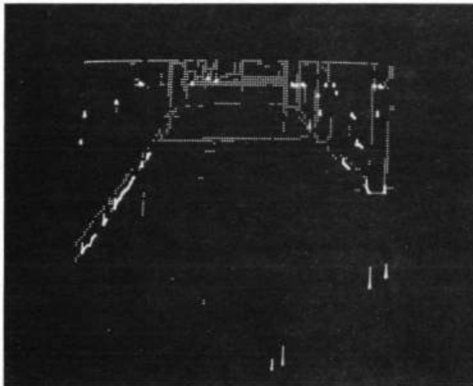
While this may seem to be a considerable amount of computation for a limited task, it is a fundamental step in the scene analysis and must be performed reliably. With the exception of problems encountered in cluttered areas, we have found this procedure to be quite reliable. Figure 1 illustrates the type of scenes that can be handled routinely. Figure 1(a) shows the view seen on the television monitor. The points found by the tracker are superimposed on a gradient picture in Figure 1(b); vertical lines mark the beginning of each segment found. The segments long enough to be kept are shown in Figure 1(c). A long-line extension of the first pseudo segment superimposed on the binary picture produced by the tracker is shown in Figure 1(d). Finally, the long lines passing the various tests are shown in Figure 1(e). Clearly, this information provides a significant start in the analysis of this scene.

C. Region Analysis

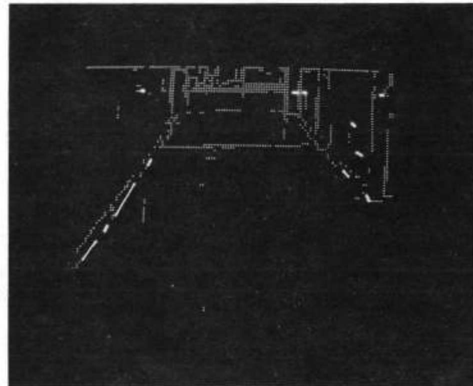
The goal of region analysis is the partitioning of a scene into regions such as wall areas, floor areas, and faces of objects, which can then be grouped and identified. The process begins by partitioning the picture into elementary regions of constant intensity. Various heuristics are used to grow these regions by merging them with their neighbors. When no further growth can be obtained by simple heuristics, the resulting regions are "described" by fitting their boundaries with straight lines, and higher level analysis begins. Some initial techniques have been investigated, [2,4] but in this area considerable work remains to be done.

Most of our recent work has involved converting basic routines originally written in LISP for the SDS-940 to more efficient assembly language versions for the PDP-10. This has resulted in quite significant reductions in the time required to process a picture, and should allow the investigation of techniques that were previously beyond serious consideration.

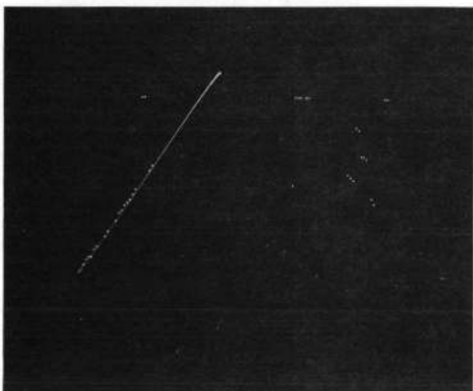
(a) MONITOR PICTURE



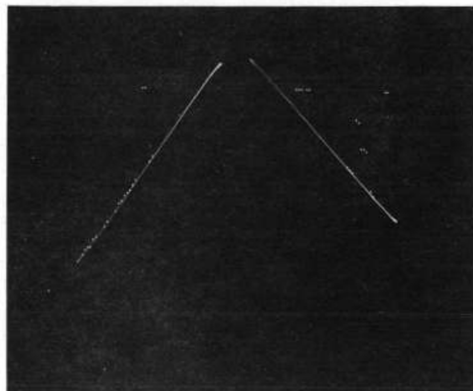
(b) TRACKED POINTS



(c) SEGMENTS



(d) FIRST LONG LINE

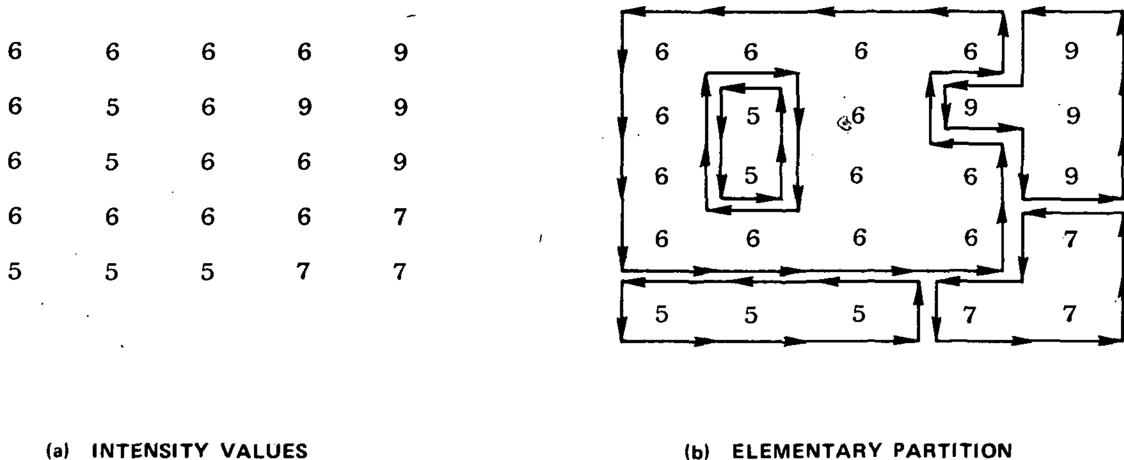


TA-8259-15

(e) LONG LINE FIT

FIGURE 1 STEPS IN BASEBOARD TRACKING AND FITTING

Three of the most basic routines are PARTITION, NEIGHBORS, and MERGE. PARTITION finds the elementary homogeneous regions in a digitized picture; its operation will be discussed below in some detail. Figure 2(a) shows a simple 5-by-5 digitized picture as an array of intensity values, and Figure 2(b) shows the 5 elementary regions that would be found by PARTITION. Note that each region is surrounded by a directed boundary. The boundary of a region may have several components, the outside component running counterclockwise and the inside components running clockwise.



TA-8259-16

FIGURE 2 ELEMENTARY REGIONS IN A DIGITIZED PICTURE

PARTITION finds these regions in two passes through the picture. During the first pass each picture element is compared with its four principal neighbors, and if a difference in intensity is found an appropriate elementary vector is inserted in the picture array. For example, if the neighbor to the right has a different intensity, a vector pointing

upward is associated with the picture element in question. At the end of the first pass a partition such as the one illustrated in Figure 2(b) is obtained.

The purpose of the second pass is to build a table that identifies the regions by number, links all of the components of a boundary, and is done by scanning the rows of the picture from left to right, starting at the top and moving to the bottom. Two marking bits are assigned to every picture element, one for its upward-pointing vector and one for its downward-pointing vector. Initially, every element is unmarked, and certain boundary elements are marked during the second pass.

As the scan proceeds, each element is inspected to see if it has either an unmarked downward-pointing vector or an unmarked upward-pointing vector. The first case corresponds to an external boundary. The scanning process is temporarily interrupted as the contour is followed and the downward-pointing and upward-pointing vectors along the boundary are marked. When the initial element is encountered again, it is assigned a special mark, a new region number is added to the region table. The specially marked element on this component of the boundary provides the link between this component and other components through the region table.

In the second case the contour is again followed and marked. This case corresponds to an interior boundary, and, while the initial element again receives a special mark and a pointer is set up to the region table, no new region number is created. Instead, the region number associated with the external contour is found by stepping to the left until a marked element is found, and by following that contour to the specially marked cell that points to the proper entry in the region table. In this way each region receives a unique number associated with its external contour. The region to which any picture element

belongs can easily be found by a similar process of stepping to the left until a boundary is encountered, following the boundary to the specially marked cell, and going to the region table.

In the initial partition of the picture, all of the elements in a region have the same intensity. Subsequent operations produce non-homogeneous regions by merging neighboring regions. However, the basic data structure describing these regions remains the same. The basic operation needed to merge two regions is merely the erasing of elementary vectors along the common boundary and the proper updating of the region table. The criteria involved for merging two regions usually involve the difference in intensity along their common boundaries. NEIGHBORS is a basic routine that finds the neighbors of a given region and computes this difference in intensity. MERGE does the erasing and updating required when two neighboring regions are merged. The coding of these basic routines in assembly language has greatly increased their efficiency, with PARTITION running more than 250 times faster than it did when written in LISP. These routines will form part of a library of subroutines that should be very useful for our future vision research.

Page Intentionally Left Blank

III THE PDP-10 COMPUTING SYSTEM

A. Status of PDP-10/PDP-15 Computer Hardware System

The system configuration is as shown in Figure 3. At present, the following subsystems are installed and operative:

PDP-10 computer

Ampex core memory

Total core: 192K words of memory

Century disk drives (including ICC controller)

Video A/D converter

We expect to complete the system by the following schedule:

PDP-15: To be delivered on 15 May

Bryant drum: To be delivered on 21 April

Adage display: To be delivered on 30 May

Second DC10B: To be delivered on 15 June

AA05B: To be delivered on 15 June

B. System Programming

The system programming that has been done in the last three months can be divided into three categories:

(1) Monitor modifications and diagnostic programs for new equipment used on the PDP-10

(2) Software for the PDP-15

(3) User programs.

The third of these categories was by far the largest. The following is a description of the programs produced in each category.

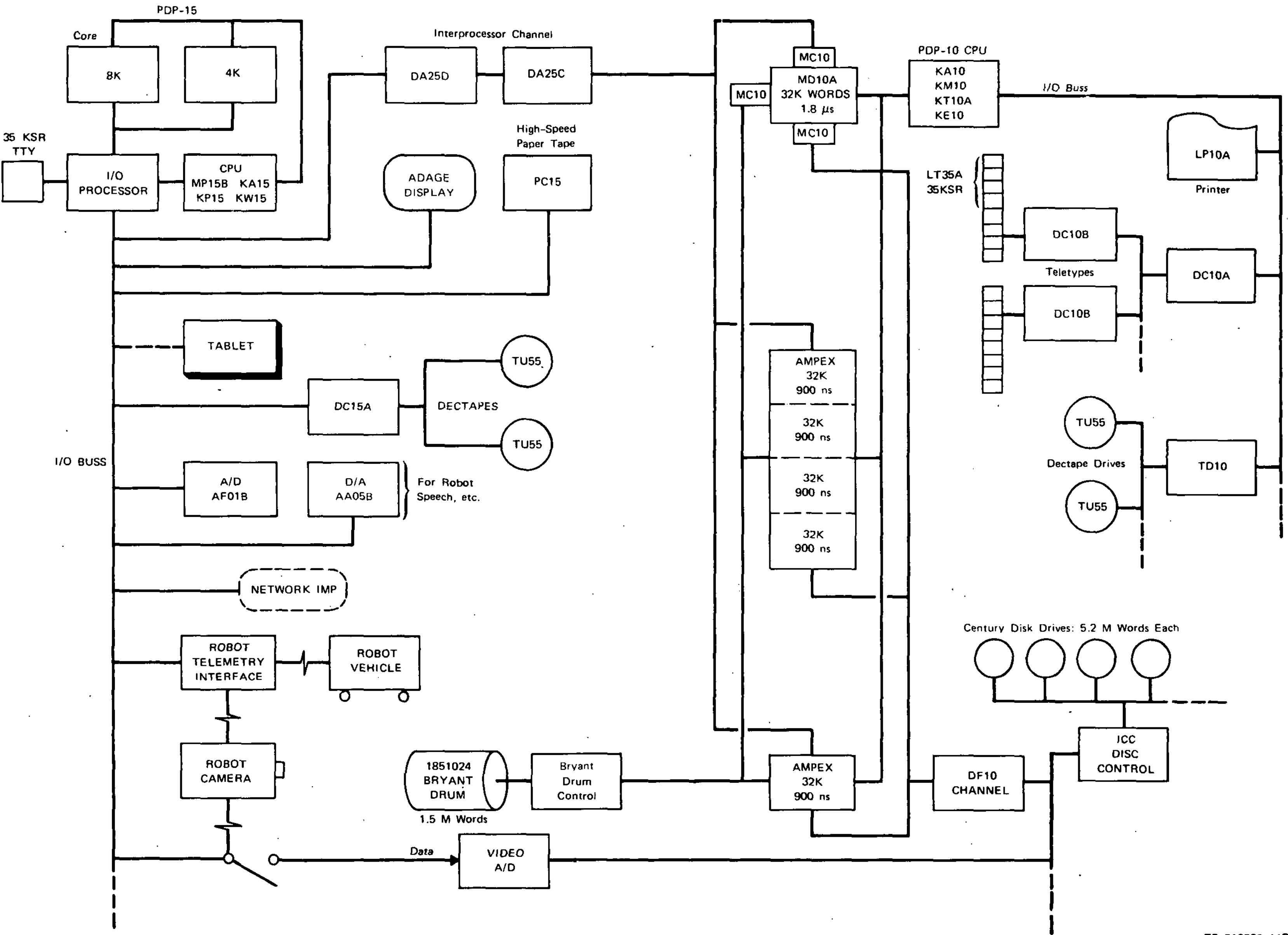


FIGURE 3 PDP-10 CONFIGURATION

1. Monitor Modifications

The first piece of equipment to require program changes to the monitor was the disk pack controller. The diagnostics were supplied (by prior agreement) by Interactive Computing Corp., who also provided the gross changes to the PDP-10 monitor. These sufficed as long as we had only two small-capacity (single density, 10 surface) disk pack drives. When we upgraded to the final configuration of four large-capacity (double density, 21 surface) disk packs, the monitor routine called DISKSER had to be changed to lengthen the STAT-BLOCK size. Also, the diagnostic had to be modified to get proper timing over the four disk packs.

The next equipment considered was the Bryant drum. Diagnostics have been written and checked out (insofar as possible without the equipment here). The modifications to the monitor (swapper and file system) are currently in progress. The diagnostic was written in a standard way, checking random patterns for parity and errors, checking timing specifications, etc.

Programs for the A/D converter to be used with the TV was incorporated into the monitor. At this time a push button simulates the robot control. The picture is entered into a temporarily created buffer, which is then written onto disk in the user's area. Any swapping is inhibited during picture receipt.

A preliminary design has been worked out for the DA25 interprocessor buffer that interfaces between the PDP-10 and PDP-15. It has not yet been incorporated into the monitor. Essentially it treats the DA25 as another device on the PDP-10's memory.

2. PDP-15 Software

In order to start work on the PDP-15 before it arrived, a simulator was written on the PDP-10. Details are described in Technical Note 25, which is included here as Appendix A. All PDP-15 work makes use of this simulator.

The first program to be checked out was the standard DEC MACRO assembler. It had to be rewritten because it depended on a DEC monitor in the PDP-15. Space and timing do not permit us to use the DEC monitor, so the assembler was modified to be self-contained. All monitor calls were changed to subroutines.

The robot drivers are being designed, as is the interrupt structure. We are making educated guesses where we lack information on PDP-15 timing.

The Adage display, which will run off of the PDP-15, has had some program design. The standard patterns were received from Adage and are being translated to PDP-15. Some other diagnostics to check timing, single-cycle operations, and PDP-15 compatibility are being written.

3. PDP-10 User Programs

A large effort was put into getting LISP to work on the PDP-10. Some storage allocation bugs were fixed. An effort to improve its facilities in the area of debugging and backtrace facilities has been started.

A LISP-to-FORTRAN interface was written. Details of this are described in Technical Note 16, which is included here as Appendix B. Also, some modifications were made so that PDP-10 binary files and FORTRAN-produced binary files would be compatible.

A set of robot programs was written for the PDP-10 (so as to use the N-tuple storage system). These generate elementary driver commands. A PDP-15 and robot simulator was written to check this out.

A routine to convert XDS 940 mag tapes to DEC-tape format was written.

Certain programming systems that were developed elsewhere have been implemented or copied on our system; among then are PILOT, FAIL, SAIL, STOPGAP II, and TREE META. Details of these languages can be found elsewhere.

The rest of the programs were all modifications to the time-sharing monitor. A new scheduler was implemented that is very similar to the one used at Stanford University written by Andy Moorer. It allows more efficient use of the PDP-10 by our users because we have large, compute-bound jobs (typically) and DEC's monitor assumes a huge amount of very small, I/O or TTY bound jobs. Along with this, a new routine to gather statistics of performance and usage was written.

The LOGIN routine was changed to eliminate passwords. The LOGOUT routine was changed to eliminate the necessity of checking all files to indicate which ones must be saved. Only those that were newly created need be checked. Most of the DEC accounting routines were eliminated.

Finally a routine that allows simultaneous access of the printer was written. This was done by creating a disk file of the print data for anyone accessing a busy printer. The monitor always scans to see if, when the printer is free, it has a file to be printed.

Page Intentionally Left Blank

IV SHORT-TERM PROBLEM SOLVING

During the period covered by this report this subgroup has been working primarily in two areas: the development of higher-level software tools for use with our new PDP-10 computer system, and the design of a new problem-solving structure that will be adequate for coping with robot tasks in the time frame of the next one to three years.

A. Software Tools

1. QA3

QA3 is a question-answering system that uses theorem proving by resolution in first-order predicate calculus as its deductive mechanism. It was developed in the SRI Artificial Intelligence Group during the past several years, and was the basis for the previous robot problem-solving system. Since it is likely that at least the theorem-proving sections of QA3 will play a major role in any new problem-solving system, the installation of QA3 on the PDP-10 has been a high-priority task. This task, which was considerably more complicated than we anticipated, is now largely complete.

a. Present Status

A skeletal version of QA3 is presently running interpretively on the PDP-10. Functions that have been debugged are those used for I/O, and most of the "operational" functions (i.e., those used in proving theorems). Since modifications and additions to the code are

being made continuously, the program will be in a state of flux for some time. Until a larger part of the code seems firm, QA3 will probably not be compiled. Very few modifications have been made that are visible to the user. Almost all changes have been syntactic (most of QA3 was coded extremely efficiently by Bob Yates using BBN LISP on the SDS 940--unfortunately, much of this code was incompatible with Stanford LISP on the PDP-10 and had to be patched).

Features that are not yet implemented in QA3 are the FILE-handling capabilities, STRATEGY options, and the facility for stopping and continuing a proof. FILES and strategies will be included in the near future, but stopping a proof requires an interrupt feature that is not yet implemented in PDP-10 LISP. (For further details about QA3, see "A User's Guide to the QA3.5 Question-Answering System," Appendix A to Ref. 2.)

b. Recent Improvements

Several changes have recently been made to make QA3 more flexible or more efficient. For example, the "quick test for subsumption" is an important time saver.

The basis of the test is the fact that a necessary (but not sufficient) condition for clause C1 to subsume clause C2 is that the set of symbols (function symbols and predicate letters) of C1 must be a subset of the set of symbols of C2. A necessary (and sufficient) condition for $A \subset B$ is that $A \cap \overline{B} = \varphi$. The implementation uses logical words (bit codes) to represent the set of symbols and checks to see if $CC1 \wedge CC2 = 0$ (where CC1 is the code for clause C1 and CC2 is the code for C2).

Every symbol in the system is assigned a bit in a word (this is that symbol's code). If there are more symbols than bits, the assignment will not be unique. This is all right (although less efficient than a unique representation) since the test is for a <u>necessary</u> condition.

The code for a clause is computed by OR'ing together the codes for all of its symbols, and this code is stored on the property list of the clause. The test function then retrieves these codes, complements the code for C2, AND's them together, and returns a value specifying whether the result is zero or not.

When a subsumption is attempted, a function is called first that computes CC1 and CC2 and returns T or NIL appropriately. Since this test is much faster than the actual subsumption calculation, it acts as a "quick filter" and results in substantial time saving.

Appendix C contains discussions of additional features that have been added to QA3 and how they may be used.

c. Future Plans

Experience with QA3 has suggested some changes and additional features that will make it a more effective system for a variety of applications including robot problem solving. Some of these changes have now been specified and partially implemented.

This next version of QA3 will include a major revision to allow experimentation with different strategies, and also to allow the theorem prover to be used in a robot planner. The revised system will contain all the executive-level features of the current one. In addition, it will be compatible with the input language for QA4 (see Section IV, "Long-Term Problem Solving") in order to allow for a possible future merge of the systems. The program will be written to allow for

easy modification (the present system is coded so tightly that it is impossible to make "clean" changes), and will be well documented to further facilitate this.

One aim of the new system will be to make it more useful for problems involving the manipulation of small, finite sets. To aid in this, two new logical quantifiers, AFS and EFS (for universal and existential quantification of variables from within a finite set) have been defined and will be added to QA3. They are defined in context as follows:

$$(AFS(x,X),\ldots,(z,Z))P(x,\ldots,z)$$

is equivalent to

$$(\forall x,\ldots,z)(x \in X \wedge \ldots \wedge z \in Z \Rightarrow P(x,\ldots,z)),$$

and similarly

$$(EFS(x,X),\ldots,(z,Z))P(x,\ldots,z)$$

is equivalent to

$$(\exists x,\ldots,z)(x \in X \wedge \ldots \wedge z \in Z \Rightarrow P(x,\ldots,z)).$$

Where x and z are variables, X and Z are the respective finite sets from which the variables will be instantiated, and P is a logical expression. These new definitions are required because first-order logic would require additional axioms to specify the finiteness (which was postulated) of the sets of instantiation variables. These axioms could take the form

$$(\forall x)(x \in X \Rightarrow x = a_1 \vee \ldots \vee x = a_n),$$

where $X = \{a_1,\ldots,a_n\}$, for each set. QA3 is unable to handle the equality relation in an efficient manner, and thus couldn't accept these axioms. In addition, these axioms require that set X (and, for that matter, Z)

be specified a priori. In fact, it will be most useful to be able to specify a form to be evaluated during the course of a proof, rather than specifying a particular set in advance.

Clauses using this new quantifier will be input to QA3 in this form

(AFS(X FORMX ... Z FORMZ) (P ... X ... Z)),

where FORMX and FORMZ are the forms that will yield the finite instantiation sets upon evaluation. The present plan is to replace a set-quantified formula by an expansion of the formula. For AFS, this expansion will be a conjunction of the appropriate instances of the original. The expansion for EFS will be a disjunction of instantiations of the original formula. In the present plan, these expansions will be made when the formula is initially encountered (probably in PRENEX). Investigations will be made into the possibility of employing a philosophy of procrastination (i.e., never do today that which can be put off to tomorrow) and only making the expansions when and to the extent that they are required.

Another feature to be added to QA3 is the ability to be called recursively. One use for this is to have the theorem prover at our disposal during evaluation of the set forms. In addition, this change will increase the overall flexibility of the system (e.g., the user could stop a proof and enter a new axiom, checking first with QA3 to see if it is inconsistent or redundant). To facilitate this new feature, bookkeeping arrays such as MEMARRAY and CLAUSEARRAY will be replaced by list structures that will be bound on entering certain functions. Each list structure will be a form of graph structure that will allow more flexibility in fetching clauses.

To allow experimentation with various strategies, the structure of QA3 will be revised. Several strategies will be available to the user, but none will be forced on him (currently UNIT PREFERENCE is an integral part of QA3). Strategies provided will be of three basic types, CHOICE strategies (e.g., UNIT PREFERENCE), FILTERING strategies (e.g., LINEAR FORMAT), and EDITING strategies (e.g., Loveland's SUBSUMPTION test). Facilities will be provided to allow the user to specify which strategies or combinations of strategies he desires. In addition, the user will be able to write his own strategy routines for evaluation by QA3. The user will have the option of revising these strategies at any time during the course of a proof. The use of a graph for memory will allow the user to assign any desired properties to a clause, and then to use these properties in conjunction with his strategies for directing his proof.

The revision of QA3 will make use of several extremely efficient routines that exist in the current system (e.g., UNIFY, RESOLVE, FACTOR, etc.). The parts that are to be rewritten are those that control these basic routines. It is expected that these changes will make QA3 even more useful as a research tool than it was previously.

2. The LISP System

The implementation of the LISP programming system, as available on the PDP-10, was far inferior (from the user's point of view) to the BBN LISP we had used on our previous (SDS-940) computer. Since LISP will be used for both QA3 and for the new problem-solving system, certain improvements were essential. One possibility is to adopt the BBN version of PDP-10 LISP. However, that system is not yet available, and will probably not run efficiently unless the BBN monitor and paging hardware

are also available. Although we are contemplating such a change, it will be at least a year away. Meanwhile, thanks largely to the efforts of Rob Kling and Jan Derksen, we have now made available a package containing good function/variable TRACE features, a BREAK capability, a BBN-type list editor, and a PRINTSTRUCTURE routine. These will make future debugging much easier. (Some of these features are described in more detail in Appendix D.)

3. Interlanguage Communication

One of the major difficulties in construction of the complete robot software system on the 940 computer was the fact that sections coded in LISP and FORTRAN could not communicate with each other except through a slow and awkward interface (called the "valet"). This was because LISP and FORTRAN each required elaborate (and incompatible) "run-time" systems, and the user machine was only 16K-words large. On the PDP-10 we have again decided to permit the use of both LISP and FORTRAN (and assembly language), partly because each of these languages is most natural for certain parts of the system, and partly because of our investment in existing code. However, the PDP-10's large core memory, long word length, and simplicity of the FORTRAN run-time system considerably reduce the interfacing problems. A powerful language interface has already been constructed by John Munson; it is described in Appendix B.

4. Source Language Flexibility

In working with expressions of first-order or higher-order logic, or descriptions of abstract operators in a planned problem-solving space, one quickly discovers the awkwardness of the standard I/O language and conventions of LISP. One would like the ability to define (and change)

the syntax of new user languages at will, and have some system automatically translate these user languages into standard internal data structures. BIP, a "Basic Interface Package," is such a system. BIP was designed by Alan Newell at Carnegie-Mellon University. It provides the builder of large programming systems a capability for easily defining notational conventions to be used for interacting with a system. Rich Fikes has implemented BIP in LISP on our PDP-10; he describes its features in Appendix E.

B. Designing a Problem-Solving System

In our past work the robot problem solver was never really designed as a system--rather, it was patched together out of components that evolved from several separate development efforts. Thus, the robot's successes in the past year were achieved by embedding a "situation calculus" into the standard QA3 system, inserting a list-structure "symbolic memory" between QA3 and the geometric grid memory used by the sensors, and building special-purpose interface functions, using the "valet," to tie everything together. Now our work has reached a level of complexity and sophistication that a true design effort seems necessary to build a viable system. The changeover to a new computer provided us with an ideal opportunity to start fresh and design a new system--of course, while keeping in mind past experience and using previously developed programs wherever possible.

This section of this report describes the current status of this design. No firm decisions have yet been made. In some areas, two or three alternative approaches are still being pursued in parallel. However, the overall structure is becoming clear and the pieces are beginning to coalesce. During the next six months we anticipate completing all design decisions and implementing the basic framework for a problem-solving system that will be useful to the project for some time to come.

1. Problem Definition

The organization of a problem solver must depend to some extent upon the class of problems to be solved. We have decided that the tasks to be given to the robot will emphasize navigation in the rooms and corridors adjacent to the present laboratory environment, and coordination of visual information with other sensory data, data in memory, and data provided "on line" manually. The emphasis will be on navigation and information-gathering tasks, rather than manipulation of objects (e.g., by pushing); however, the new system must be capable of at least the "collect" task accomplished by the previous system. The first new major task to test the robot will be, "Go down the hall and stop in front of every open doorway." The primary consideration in constructing a solution to this task will be generality in the structure, so that we learn how to organize the solutions to future, more complex, corridor tasks. Error correction and responding appropriately to unexpected or uncertain information are important abilities that have been ignored in the past but must be provided in the new system.

The principal elements of the new system will be the model, the operators, a planner, and an executive.

2. The Model

The robot's model of the world will be based upon the n-tuple storage system embedded in LISP (see "The N-Tuple Storage System," Appendix C to Ref. 2). However, simple "tuples" (short for "n-tuples") of symbols cannot provide enough information in a natural way. Therefore, the model will have the following kinds of elements:

(a) Tuples correspond to ground predicates. Examples are:

(IN ROBOT G23)

(TYPE G23 ROOM)

(NAME G23 K2070)

These will be stored directly in the tuple memory.

(b) Special Information Structures

1. Grids

2. Special routines for evaluating predicates.

These will be accessed by pointers stored in tuples.

(c) Predicate Calculus Statements that are valid in all models (eternal truths). Examples are:

(1) (TYPE G24 DOOR) ∧ (TYPE G25 DOOR) ∧ [(OPEN G24) ∧ (OPEN G25)]
(At least one of doors G24, G25 is always open.)

(2) (∃x)[(TYPE x DOOR) ∧ (OPEN x)]
(There is at least one open door.)

(3) (∀x)(∃y)(∃z)[(TYPE x ROOM) ∧ (TYPE z ROOM) ∧ (TYPE y DOOR) ∧ (CONNECTS x y z)]
(Every room has at least one doorway into another room.)

(d) State-Dependent Nonatomic Formulas (These might, for example, be sentences derived from categories 1 and 3, or may be supplied by a sensory system or by the experimenter.) Examples are:

From the "eternal truth":

(IN ROBOT G23) ⇒ (HOME ROBOT)

and the N-tuple:

(IN ROBOT G23)

we can deduce:

(HOME ROBOT) .

Note that (HOME ROBOT) is state-dependent and must be eliminated from the model whenever (IN ROBOT G23) ceases to be true. We shall provide a special mechanism for eliminating all state-dependent formulas when their elimination is appropriate.

(e) Special tuples

There will be some special tuples whose purpose will be to store relevant information from the past and to act as flags. Examples are:

(BEENTHERE G24) (BEENTHERE G25)	Stores the facts that the robot has visited G24 and G25.
(PICTURETAKINGFLAGSET)	Stores the fact that the robot is ready to take a picture.

3. Operators

The operators are routines that can be executed by the Executive to accomplish some action. Special documentation (available to the planner) called operator descriptions gives information about

(a) Under what conditions they can be applied

(b) Their expected effects

(c) Their reliability

(d) Their "cost."

The planning system uses this information in constructing a plan (which is just a sequence of operators).

Some typical operators might be:

(1)	GOTO ($\vec{x}$)	Takes robot to point $\vec{x}$ if $\vec{x}$ is in the same room as the robot.
(2)	GOTHRUDOOR (x)	Takes robot thru doorway x.
(3)	GOTOADJRM (x)	Takes robot to the adjacent room x.
(4)	ISDOOROPEN (x)	Finds out whether door x is open.
(5)	CLEARPATH ($\vec{x}$)	Finds out whether there is a clear path to point $\vec{x}$.

In all of these, the operator is really an operator <u>family</u> identified by a particular name. The value of the parameters or "arguments" of the operator determines the specific action within the family.

Since the action of an operator may change the state of the world, part of each operator's description must specify what features of the model that operator affects.

The operators may be viewed as a set of parameterized algorithms that when executed will cause the robot to perform some specific activity (e.g., move x inches forward, take a television picture, go to adjacent room) and/or will make some specified change in the system's model of the world. The basic system will contain a set of <u>primitive operators</u> such as move x inches forward, turn x degrees, tilt the camera x degrees, etc. Operators that accomplish more complex operations (such as go to adjacent room x) will be written using these primitive operators.

We wish to design the system so that when a new class of problems is considered, it will be relatively easy to define additional operators in the system. This facility will allow us to provide the

system with the basic operations needed to solve problems in any given problem domain.

Since the system creates plans consisting of sequences of operators, it is necessary for the planner to have information describing the operators available in the system. In particular, it needs to know under what circumstances an operator can be successfully executed (e.g., a "go through door x" operator can be executed only if door x is open), what the range of possible arguments to the operator is (e.g., a "turn x degrees" operator might restrict x to be an integer in the range $-360 \leq x \leq 360$), what the results of executing the operator will be (e.g., a "go to adjacent room x" operator would change the location and orientation of the robot such that the robot's new location is in room x), etc. We have proposed an operator definition language with which the person who is adding a new operator to the system can easily provide the information needed by the planner. The system's interpreter for this language would be able to accept an operator definition in this language and then use the new operator during planning on an equal basis with all the other operators defined in the system.

A Backus Naur Form description of the operation definition language is as follows:

```
⟨operator definition⟩  ::=

     operator ⟨operator name⟩  (⟨argument⟩,...,⟨argument⟩);

          begin ⟨declaration⟩;...;⟨declaration⟩;

               initial conditions
                    ⟨condition⟩;...;⟨condition⟩;

               tuple transformations
                    ⟨transformation⟩;...;⟨transformation⟩;

               resulting conditions
                    ⟨condition⟩;...;⟨condition⟩;

          end;
⟨operator name⟩ ::= ⟨identifier⟩
⟨argument⟩ ::= ⟨identifier⟩
⟨declaration⟩ ::= ⟨type⟩⟨identifier⟩,...,⟨identifier⟩
⟨type⟩ ::= ⟨integer⟩ | ⟨symbol⟩ | door | face | room | object | sign
⟨transformation⟩ ::= ⟨old tuple⟩ → ⟨ new tuple⟩ | ⟨old tuple⟩ → ⟨ new tuple⟩
⟨old tuple⟩ ::= ⟨$tuple⟩

⟨$tuple⟩ ::= "⟨" ⟨$element⟩,...,⟨$element⟩ "⟩"
⟨$element⟩ ::= $ | ⟨symbol expression⟩ | ⟨integer expression⟩

⟨new tuple⟩ ::=  ⟨?tuple⟩

⟨?tuple⟩ ::= "⟨" ⟨?element⟩,...,⟨?element⟩ "⟩"
⟨?element⟩ ::= ? | ⟨symbol expression⟩ | ⟨integer expression⟩
⟨condition⟩ ::= ⟨boolean expression⟩    .
```

An operator definition begins with the name of the operator followed by its list of arguments. The Boolean expressions on the initial conditions section of the definition are interpreted as a conjunction that must be true before the operator can be executed. The conditions define the domain of arguments and robot world models over which the operator is defined.

The set of operators and operand types that we will allow in the condition expressions has not yet been specified, but our goal is to have the system accept as rich a class of expressions as possible. We would like the language to include standard arithmetic operators over the integers (e.g., plus, minus, times, divide, equality), standard logical connectives and quantifiers (e.g., conjunction, disjunction, implies, for all, there exists, negation), and standard set operators (e.g., union, intersection, compliment). The constraint of the inclusion of these desirable features into the language is our ability to design a deductive system that can effectively deal with expressions in the language. Hence, the condition language will continue to expand as the power of our problem-solving and theorem-proving programs increases.

We have defined the function element and the predicate inmodel in the condition language to facilitate the fetching of data from the robot's world model and the testing for the occurrence of particular tuples in the model. The predicate inmodel has the following form:

inmodel(⟨$tuple⟩) .

The value of inmodel is true if at least one tuple matching the $tuple argument can be found in the model, and false otherwise. A $tuple is matched to the tuples in the model by first evaluating each element of the $tuple and then doing a fetch operation to determine if the evaluated tuple exists in the model. The atom "$" is interpreted to match with anything during the fetch so that any tuple found in the model that matches the non$ elements of the evaluated tuple will satisfy the fetch and make inmodel true. For example, inmodel(⟨joinsrooms,$,K200,K235⟩) is true only if there is at least one door defined in the model that joins rooms K200 and K235.

The function *element* has the following form:

element(⟨integer⟩,⟨$tuple⟩) .

The value of *element* is determined by first fetching all tuples from the robot's world model that match the $tuple argument. The atom "$" is assumed to match with anything during the fetch as described above. If the first argument of *element* is the integer k, then the value of the function is the set formed by the kth elements of each of the fetched tuples. For example, *element*(2,⟨inroom,$,K200⟩) denotes the set of all objects defined in the model to be in room K200.

The tuple transformations section of the definition indicates which tuples in the robot's world model are changed, added, or deleted by a successful execution of the operator. The transformations are interpreted to mean that each *old tuple* is removed from the robot's model and each *new tuple* is added to the model by the operator. The occurrence of "?" in a *new tuple* indicates a tuple element whose value cannot be specified in the operator description. For example, a "go to adjacent room x" operator will change the orientation of the robot, but the orientation resulting from any particular execution of the operator cannot easily be described; hence a transformation containing the *new tuple* "⟨theta,robot,?⟩" could be used to indicate the unpredictable change.

The tuple transformations also provide additional initial conditions in that for every *old tuple* that occurs in a transformation, *inmodel* of that tuple must be true for the operator to be applicable.

The Boolean expressions in the resulting conditions section of the definition are interpreted as a conjunction that will be true after successful execution of the operator. These conditions combined with the *new tuples* in the tuple transformation define the range of robot world model produced by successful execution of the operator.

The confidence section of the definition contains an integer expression whose value provides an indication of the probability that execution of the operator will be successful. For example, the probability that a "move forward x inches" operator will succeed when the robot is in a room r1 might be an expression involving x, the size of room r1, the number of objects in room r1, and the amount of unknown area in room r1. The confidence expression is assumed to have value x in the range $0 \leq x \leq 100$, where 0 is assumed to be no chance of succeeding and 100 is certainty.

The cost section of the definition contains an integer expression whose value provides an indication of the expected cost of executing the operator. The expected cost is a combination of the amount of expected computer processor time and the expected time required for the robot to carry out the activity specified by the operator.

The body section of the definition is the actual LISP program that is executed when the executive invokes the operator.

The declarations that occur at the beginning of the operator definition provide a list of "local variables" for the definition. These local variables may be thought of as being existentially quantified over the entire definition (except for the body) in the following manner: Given that x1, x2,...,xn,y1,y2,...,ym occur in the declarations, that which xi occurs in the initial condition or in an <u>old tuple</u>, and that none of the yi occurs in the initial conditions or in an <u>old $tuple</u>, then the definition states that if there exist values for x1,x2,...,xn such that the initial conditions are true and each <u>old tuple</u> occurs in the robot's world model, then there exist values for y1,y2,...,ym such that for the same values of x1,x2,...,xn a successful execution of the operator will perform the indicated tuple transformations and produce

the resulting conditions at the expected cost indicated by the cost expression with the confidence indicated by the confidence expression.

The following is an example of an operator description for the system: operator defdoor(direction);

```
begin integer x,y; face f1,f2,f3; door d1;

        initial conditions
            direction in {eastface,westface,northface,southface};
            f1 = element(3,⟨direction,element(3,⟨inroom,robot,$⟩),$⟩);

        tuple transformations
            ⟨grid,f1,x,y,unknown⟩ → ⟨grid,f1,x,y,known⟩;
            ⟨pan,robot,$⟩ → ⟨pan,robot,?⟩;
            ⟨tilt,robot,$⟩ → ⟨tilt,robot,?⟩;
            → ⟨doorlocs,d1,?,?⟩;
            → ⟨type,d1,door⟩;
            → ⟨joinsfaces,d1,f2,f3⟩;

        final conditions
            f2 = f1 ∨ f3 = f1;

        confidence
            ---

        cost
            ---

        body ---

end;
```

The function of the operator being described is to take a picture of one of the wall faces of the room that the robot is in and define in the model any doors found on that wall from the picture. The wall face of interest is specified to the operator by the argument "direction." The declarations indicate that six existentially quantified variables will occur in the definition--two integer values, three

wall face names, and one door name. The initial conditions for the operator require that "direction" be one of four names, and that f1 will denote the name of the wall face in which doors are to be found. The first old tuple in the tuple transformations indicates that there must exist some point (x,y) on wall face f1 that is marked unknown. This requirement assures that the operator will be applied only to wall faces that have not been previously completely explored. The tuple transformations indicate that the pan and tilt angle of the robot's camera will be changed to unknown positions, that the point (x,y) on wall face f1 become known in the model, and that three tuples associated with a newly defined door d1 will be added to the model. These transformations imply that an execution of this operator is considered successful only if at least one new door is found in the wall face. The resulting conditions indicate that one of the wall faces that door d1 connects must be f1.

The confidence expression for this operator is a measure of the probability that a new door will be found in the picture. It could be expressed as a function of the amount of unknown area on that portion of the wall face that is in the range of the robot's camera. The cost for this operator could be expressed as a function of the cost of an average pan and tilt of the camera, the cost of taking and processing a picture, and the cost of entering the new information into the robot's model.

4. The Planner

The heart of the problem-solving system is the planner--the program that specifies the sequence of operators to be invoked to accomplish each specified task.

a. Deductive Mechanisms

The planner must have some deductive mechanism to see if a hypothetical model (resulting from applying an operator) satisfies the goal conditions. Also it must be able to see if a model satisfies preconditions for selecting an operator. Possibly QA3, with major additions enabling efficient handling of sets, equality, integer arithmetic, etc., should be used here. At first, we will probably restrict the tasks so that complex deductions need not be made for goal and precondition testing. (That is, if a model satisfies a goal or precondition, it will satisfy it in an unsubtle, obvious manner.)

For some tasks, the original goal predicate (as provided by the experimenter or by an English-logic translator) can be simplified considerably resulting in more efficient problem-solving. Consider, for example, the following task:

"Visit all of the known offices whose doors are open."

For this task, we use the special n-tuple, (BEENTHERE ...) .

In predict calculus, the task could be stated as

$$(\forall x)[(\text{TYPE } x \text{ OFFICE}) \wedge (\text{DOORSTATUS } x \text{ OPEN}) \Rightarrow (\text{BEENTHERE } x)]$$

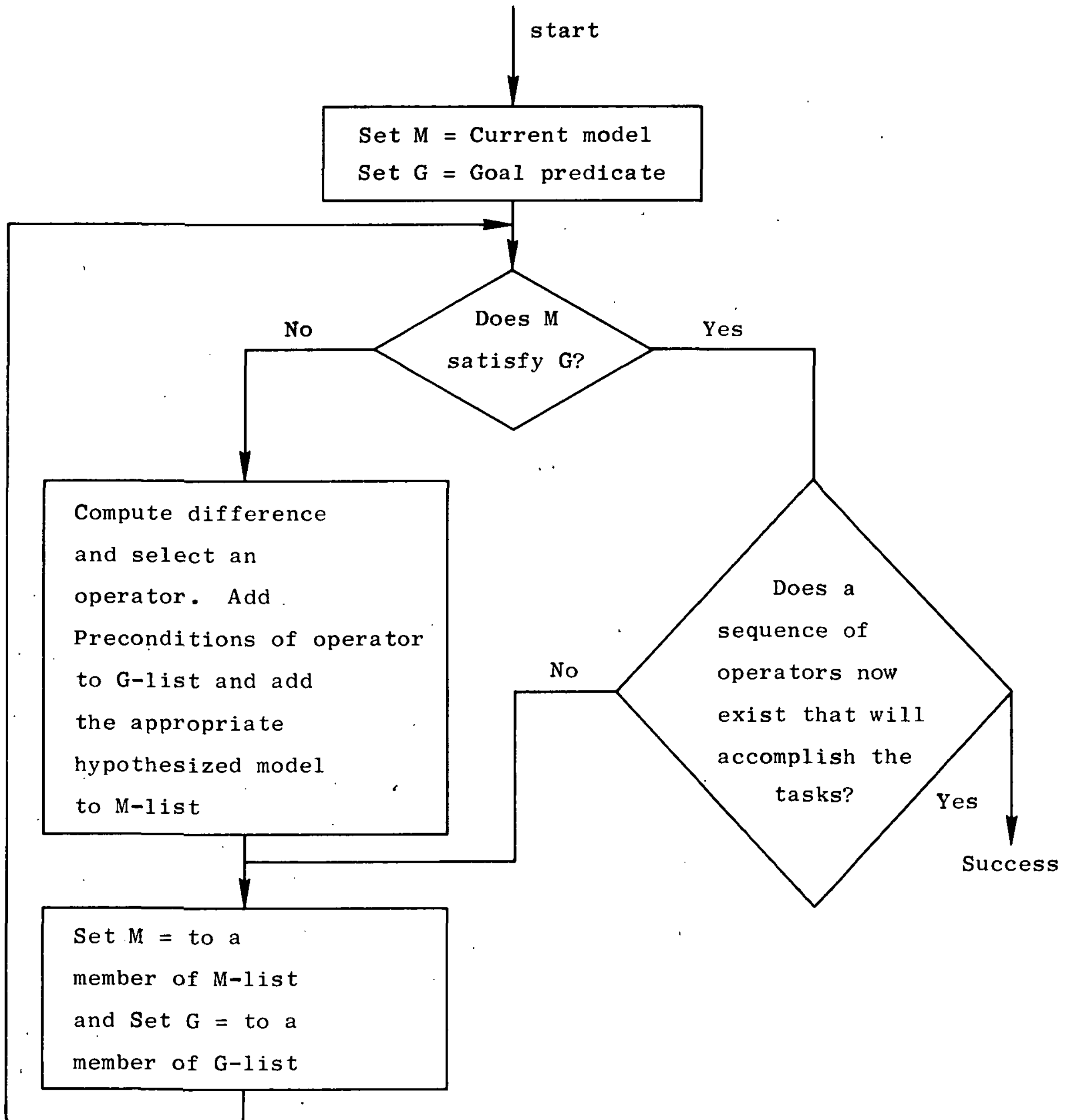
If a special deductive mechanism has access to the current model and to information about the effects of operators, it can substantially simplify this statement. For example, suppose no operator affects the tuples TYPE and DOORSTATUS and suppose in the current model there is just one office whose door is open, namely G23. Then an equivalent, simpler, task statement is:

(BEENTHERE G23) .

The finite set quantifiers AFS and EFS, discussed in Section IV-A-1c, will provide the ability to make some of these simplifications in QA3.

b. Planner Organization

The specific structure of the planner has not yet been decided. One possibility is to construct a GPS-like planner that attemps to construct a sequence of operators to achieve the goal. The operation of such a planner is roughly described by the following chart:



An attractive alternative is to encode the operators into clauses of predicate calculus and then use QA3 itself as the planner. QA3 would have to be modified to permit the insertion of special operator-selection heuristics in appropriate places--but, in the absence of such special heuristics, the normal theorem-proving heuristics could take over. Both these approaches are currently being explored and may be combined.

c. Planner Bookkeeping

Any planner must be able to consider the effects of hypothetical actions. Therefore it must "grow" a tree of possible models, and keep track of the relations among them. This results in surprisingly complicated bookkeeping problems. "Model schema"--parameterized classes of models--will play an important role in this bookkeeping, by permitting the automatic selection of particular models from classes of similar ones. Consider the following example (assuming a predicate calculus representation for models):

(1) Let $M(x)$ be a model schema. $M(x)$ is a set of clause schemas containing the variable, x. Note: x is not one of the variables that is universally quantified in the clauses; it is merely a variable of the schema. That is, if $x\sigma$ is an instance of x, then $M(x\sigma)$ is a particular model that is an instance of the schema.

(2) Let $C(x)$ and $C'(x)$ be two clause schemas in $M(x)$ such that there exists a most general instance $x\lambda$ of x permitting $C(x\lambda)$ and $C'(x\lambda)$ to possess a resolvent $R(C,C')$. There may be more than one resolvent, and thus there may be more than one λ. Note: For the purposes of finding a λ, we can temporarily regard x as an ordinary (universally quantified) variable of C and C'. Then, if C and C' possess a resolvent using a most general unifier τ, that part of τ specifying the substitution for x is a permissible λ.

(3) For any such λ, we can produce an inferred instance of the model schema $M(x)$. The instance M' is given by

$$M' = M(x\lambda) \cup R(C,C') \quad .$$

(4) This operation of obtaining an inferred instance of a model schema can be explained as follows: $M(x)$ certainly has an instance $M'' = M(x\lambda)$. Furthermore $M(x\lambda)$ has clauses $C(x\lambda)$ and $C'(x\lambda)$ possessing the resolvent $R(C,C')$. We can consider the resolution actually to occur in the model instance M''. Adding the resolvent to M'' then produces the model instance M'. For convenience, however, we can perform the resolution directly in M (allowing x to be substituted for) if we are careful to substitute for x throughout M.

(5) More specifically: Suppose the model M_o consisting of the clauses

$$\begin{array}{l} AT(robot,R) \\ AT(OB1,a) \\ \sim AT(OB1,b) \vee G \end{array} \quad .$$

From these clauses we can produce the model schema $M(x)$ by using the push (OB1,x) operator. $M(x)$ is

$$\begin{array}{l} AT(robot,x) \\ AT(OB1,x) \\ \sim AT(OB1,b) \vee G \end{array} \quad .$$

Now we can obtain an inferred model instance M' by considering x to be a variable that can be substituted for in resolution. M' is

$$M(x_{(x,b)}) \cup G$$

or

$$M' = \left\{ \begin{array}{l} AT(robot,b) \\ AT(OB1,b) \\ \sim AT(OB1,b) \vee G \\ G \end{array} \right\} \quad .$$

5. The Executive

Finally, it will be the executive's responsibility to handle a variety of items such as decisions to abandon planning and start executing, decisions to gather more information, supervising the execution of operator sequences, and making certain changes to the current model after operator execution. The executive should also be concerned with the "cost effectiveness" of the performance of the whole system. (See Appendix E for an extensive discussion of this important issue.)

The executive is the least-understood portion of the problem-solving system at the present time. We expect to start by implementing a trivially simple "mark zero" executive, and then gradually evolving to more sophisticated versions.

a. Mark Zero Executive

This system will work if the current model is a completely adequate representation of the world and if the execution of operators is never interrupted by unexpected bumps, etc. It simply executes the operators proposed by the planner, in sequence, and "gives up" with a message on the teletype if for any reason it cannot continue.

b. Mark 1 Executive

This system assumes that the planner uses a model thought to be complete, but that unexpected interrupts may occur during execution of an operator. Of course, the occurrence of such interrupts may add new information to the current model, but usually the model is not changed in the way that it would be if the operator were expected normally. Therefore the precondition for applying the next operator in the sequence may not be met. The Mark 1 executive will test for this possibility before applying any operator by checking to see if the preconditions

of the operator are met by the current model. If they are, it continues normally. If not, it recalls the planner section. (The planner may remember the results of previous searches as well as the remaining unexecuted operator sequence.)

c. Sensory Verification after Operator Execution

The effects of certain operators are known only approximately. After execution of such operators we cannot be sure that the world is precisely as the model represents. Therefore there is an obvious need to check the world with various sensory equipment to see if the model adequately describes the world.

We could take the view that these sensory checks are special "information-gathering" operators that the executive automatically inserts after certain of the action operators. Examples of possible operators of this type are:

(1) Find the position of the robot with respect to some object

(2) Find the position of a wall

(3) Find the position of a door in a wall

(4) Determine whether a door is open

(5) Read simple messages

(6) Locate and classify objects in a scene

(7) Conduct a "complete scene analysis" .

Alternatively, the sensory checks could be accomplished implicitly by specialized action operators whose execution accomplishes certain subtasks highly reliably (most probably by employing sensory feedback).

Example: The following sequence of operators takes the robot from one room into an adjacent room:

GOTO $(\vec{x})$

TURN (α)

GOTHRUDOORWAY .

Because of accumulated error, we should have a position and orientation reading after execution of the TURN (α) and before execution of the GOTHRUDOORWAY operator. The object of such a reading would be to ensure that the robot is adequately aligned with the doorway. These checks could be automatically planned by the problem solver if we require as a precondition of GOTHRUDOORWAY that a "ready flag" had been set in the model and provide an operator, READY, that "readies" the robot for going through the doorway. The effect of READY on the model is to set the ready flag by inserting a tuple (READY). (Any subsequent motion deletes this tuple.) Execution of READY involves picture-taking (say) and feedback-directed motion to align the robot in front of the doorway. We then also specify that a precondition for applying READY be that the robot's position in the model be at the appropriate place. With these added features, the planner itself can be used to plan for certain sensory readings.

d. Mark 2 Executive

This system will be able to call for the gathering of new information whenever planning is frustrated by an incomplete model.

If the planner fails to find a list of action operators for achieving the goal, it is because the goal is unachievable with the present model. The Mark 2 executive will be designed to analyze the reason for this failure and match an appropriate information-gathering operator to this reason. After successful execution of this operator,

and with a more informed model, the system reenters planner to find a plan to achieve the original goal.

The Mark 2 executive will also need some "cost-effectiveness" mechanism (see Appendix F) for monitoring the progress of the planner. Only when the planner is unable (by the cost-effectiveness criteria) to generate an acceptable plan of action operators, are information-gathering operators considered.

Page Intentionally Left Blank

V LONG-TERM PROBLEM SOLVING

A. Introduction

The long-term problem-solving effort has been involved with the design and implementation of a general-purpose, formal problem-solving system. The current system, termed QA4, is based upon mechanized theorem proving in higher-order logic and emphasizes the role of semantic information and flexible control strategies. Two major applications of such a system are in the field of automatic program writing and in robot planning and problem solving.

The QA4 system represents a significant extension and modification to the ideas incorporated in a resolution-based first-order theorem prover such as QA3.5. First the system is intended to be semantically rather than syntactically oriented. The methods or procedures used by the system to solve a particular problem should be related to the semantics of that problem and not applied uniformly to all problems. Secondly, QA4 is embedded in the formal framework of ω-order predicate calculus. This provides a much richer language than first order and permits the syntax and semantics of any portion of the system to be expressed in its own language. Moreover the addition of sets to the language provides a powerful interface between logical operations and computational operations (such as set enumeration etc.). Finally, the QA4 strategies will be user-defined and expressed directly in the QA4 language rather than being "frozen into" the problem-solver's code.

The following discussion is a description of QA4 at its present state of development.

B. The Logic Language

The current QA4 language is an extension of higher-order logic as defined by Robinson.[6] Set, bag, and tuple expressions and operations have been added, and the bound variable occurring in LAMBDA expressions has been significantly modified.

1. The Class of Expressions

A QA4 expression falls in one of seven syntactic categories: identifies, numbers, applications, set expressions, tuple expressions, bag expressions, and bound-variable expressions.

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, UNION, etc. are called special in that they have predetermined, built-in meanings. All other identifiers are called nonspecial and may generally be used for variables, defined functions, etc.

Applications. An application is an expression of the form F(A) [alternatively (FA) or FA] 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 ⟨A1,A2,A3...⟩ so there is no loss of generality. The meaning of an application F(A)is its natural one--namely the result of applying the function (denoted by) F to the argument (denoted by) A. QA4 has an infix language that will be used in the remainder of the discussion. If one writes A ⇒ B, the symbol ⇒ is an infix operator and the expression is translated into the application IMPLIES ⟨A,B⟩ where ⟨A,B⟩ is the single argument of IMPLIES. Similarly 3 + X + Y is translated into PLUS[3,X,Y] where [3,X,Y] is a bag (discussed below). Thus the infix expression should be understood as an abbreviation of a corresponding application.

Sets. A *set expression* is an expression of the form

$\{E_1,\ldots,E_n\}$ (the QA4 parser uses [: for {, and :] for }),

where $E_1,\ldots,E_n$ are any expressions. The meaning of the set expression $\{E_1,\ldots,E_n\}$ is the set of objects denoted by $E_1,\ldots,E_n$. Since the order of the elements $E_1,\ldots,E_n$ in a set is immaterial--as well as multiple occurrences of elements--the sets

$$\{A,B,C\} \quad \{C,A,B\} \quad \text{and} \quad \{C,A,A,B,C\}$$

are treated as identical expressions.

Tuples. A *tuple expression* is an expression of the form

$$\langle E_1,\ldots,E_n\rangle \quad ,$$

where $E_1,\ldots,E_n$ are expressions. Its meaning is the n-tuple $\langle \bar{E}_1,\ldots,\bar{E}_n\rangle$ of objects, where E_i denotes the object $\bar{E}_i$. Two tuples $\langle E_1,\ldots,E_n\rangle$ and $(F_1,\ldots,F_m)$ are logically equal provided they have the same length $(n=m)$ and each E_i is logically equal to F_i. So we have $\langle 3+1,2-1\rangle = \langle 4,1\rangle \neq \langle 1,4\rangle$. Tuples are used as arguments to functions generally demanding more than one argument. Thus a function $f(x,y)$ in mathematics would be represente in QA4 as a function $F\langle X,Y\rangle$ where F takes the tuple $\langle X,Y\rangle$ as its single argument.

Bags. A bag expression is an expression of the form $[E_1,\ldots,E_n]$, where $E_1,\ldots,E_n$ are arbitrary expressions. A bag is like a set except that elements may have multiple occurrences. For example, $[2,3,2] = [2,2,3] \neq [2,3]$. Bags are used as arguments to functions such as + (plus) and * (times) that are commutative and associative. Thus $7 = +[2,2,3]$.

Numbers. A number is simply a positive or negative integer at present; e.g., 3, 0, -2 are numbers.

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, and *keyword* is one of the following: LAMBDA, FORALL, EXISTS, CHOICE, ... So all LAMBDA expressions and quantified expressions are bound variable expressions. The bound variable, BV, has its own syntax--and semantics that extend the usual definition. It is more akin to a variable declaration. Basically, the purpose of a bound variable is to assign values to one or more variables for a temporary duration--namely for the evaluation or analysis of the expression E, the body of a bound variable expression.

A *complete* bound variable is a triple (name structure predicate) where *name* is an identifier, *structure* is a tuple, bag, or set of BVs (or a decomposition structure), and *predicate* is any truth-valued expression. When the bound variable is bound to an expression, the name is set equal to the expression, the expression is decomposed if possible according to the structure, and the predicate is tested on the results of the decomposition.

For example, (X ⟨Y,Z⟩ Y+Z<5) is a bound variable having X as its name, ⟨Y,Z⟩ as its structure, and Y+Z<5 as its predicate. If we attempt to bind this bound variable to the expression ⟨3,1⟩, we set X = ⟨3,1⟩ and then bind ⟨Y,Z⟩ to ⟨3,1⟩ by setting Y=3 and Z=1. Then Y+Z<5 is tested and found *true*, so the binding succeeds.

More precisely, after assigning the name to the expression, the structure (if present) is examined. If the structure is a tuple (of bound variables), then the expression must itself be (or denote) a tuple of the same length. Then the elements of this structure are recursively bound to the corresponding elements of the expression.

If the structure is a set or bag of bound variables, then again the expression must be a set or bag respectively and the bound variables of the set (or bag) structure are bound to elements of the expression.

Here the situation is more complex since there is more than one possible assignment. The predicate is used to restrict the possible assignments to the one desired.

For example, to bind (S {X,Y,Z} X<Y & Y<Z) to the set {3,1,2} the only possible assignment is X=1, Y=2, Z=3, S={3,1,2}.

One further structure is the structure decomposition that takes the form BV1 · BV2 or BV1 :: BV2. The dot and double colon are termed "cons" and "append" decompositions respectively. To illustrate how the binding takes place in this case we give an example. Bind (T X · T1) to ⟨3,2,4⟩. X is set to 3, T1 is set to ⟨2,4⟩, and T to ⟨3,2,4⟩. Thus X · T1 decomposes the tuple ⟨3,2,4⟩ into its first element 3 and the remainder ⟨2,4⟩. (The absence of the predicate means simply that no predicate test is made.)

An example of the append decomposition is BIND (X Y :: Z length (Y) = 2) to ⟨3,2,4⟩ assigns X = ⟨3,2,4⟩, Y = ⟨3,2⟩, and Z = ⟨4⟩. The elements Y and Z must both be tuples, Y of length 2 and Z arbitrary; Y and Z appended together must yield the bound expression.

<u>LAMBDA Expressions</u>. A LAMBDA expression is a bound variable expression of the 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 ⟨X,Y⟩, where ⟨X,Y⟩ is a function that revises a tuple of length 2. ⟨X,Y⟩ is the BV, and ⟨Y,X⟩ is the expression body.

<u>Quantified Expressions</u>. A quantified expression is an expression of the form (quantifier BV E), where E is truth-valued and quantifier is either the universal quantifier FORALL or the existential EXISTS. The mathematical statement $\forall_x \forall_y \forall_z P(xyz)$ is expressed in QA4 as (FORALL ⟨X,Y,Z⟩ P⟨X,Y,Z⟩). (Similarly for EXISTS.)

2. Primitive Operations

The primitive QA4 operations can be broadly separated into three categories: logical operations, set and tuple operations, and arithmetic operations. The following list gives most of the basic operators.

Logical Operators

AND$\{P_1, \ldots, P_n\}$

The operator AND takes a set of truth values and returns true if the set consists of the singleton {true} and false otherwise. Thus AND$\{P_1, \ldots, P_n\}$ is true provided all the expressions $P_1, \ldots, P_n$ denote true. In the infix language we could write P_1 & P_2 & ... & P_n.

OR$\{P_1, \ldots, P_n\}$

OR is analogous to AND except that it returns true if true is a member of the set and false otherwise. In the infix language we have $P_1 \vee P_2 \vee \ldots \vee P_n$.

Since AND and OR are both commutative and associative, they have been made into a set as an argument rather than a tuple.

EQUAL$\{E_1, \ldots, E_n\}$

EQUAL asserts that all of the members of the set are logically equal--and denote the same element. In the infix language we have $E_1 = E_2 = \ldots = E_n$.

NOT(P)

NOT negates the truth value of its argument. In the infix language we have #P.

IMPLIES$\langle P_1, P_2 \rangle$

IMPLIES asserts that P_1 implies P_2.

Infix: $P_1 \Rightarrow P_2$.

Set, Bag and Tuple Operations

IN$\langle X,S \rangle$ asserts that X is an element of the set S.

Infix: X IN S.

UNION$\{S_1, S_2, \ldots, S_n\}$ is the set-theoretic union of the sets $S_1, \ldots, S_n$.

INTERSECTION$\{S_1, \ldots, S_n\}$ is the set-theoretic intersection of the sets $S_1, \ldots, S_n$.

DIFFERENCE$\langle S_1, S_2 \rangle$ is the set-theoretic difference $S_1 \sim S_2$ of the sets S_1 and S_2.

APPEND$\langle T_1, \ldots, T_n \rangle$ adjoins the tuples (or bags) $T_1, \ldots, T_n$ so if $T_1 = \langle e_i^1, \ldots, e_{m_1}^1 \rangle, \ldots, T_n = \langle e_1^n, \ldots, e_{m_n}^n \rangle$ then

APPEND$\langle T_1, \ldots, T_n \rangle = \langle e_1^1, \ldots, e_{m_1}^1, e_1^2, \ldots, \ldots, e_{m_n}^n \rangle$.

Infix: $T_1 :: T_2 :: \ldots :: T_n$.

ADD$\langle X,T \rangle$ adjoins the element X to the tuple (or bag) T. If $T = \langle X_1, \ldots, X_n \rangle$ then ADD$\langle X,T \rangle = \langle X, X_1, \ldots, X_n \rangle$.

NTH$\langle T,n \rangle$ extracts the nth element from a tuple T. If $T = \langle X_1, \ldots, X_n, \ldots, X_m \rangle$ $m \geq n$, then NTH$\langle T,n \rangle = X_n$.

Arithmetic Operations

PLUS$[n_1, \ldots, n_m]$ forms the sum of the elements $n_1, \ldots, n_m$ in the bag argument.

Infix: $n_1 + n_2 + \ldots + n_m$.

TIMES$[n_1, \ldots, n_m]$ forms the product $n_1 \cdot n_2 \ldots n_m$.

Infix: $n_1 * n_2 * \ldots * n_m$.

GT⟨m,n⟩ asserts that m is greater than n.

Infix: m >· n.

LT⟨m,n⟩ asserts that m is less than n.

Infix: m <· n.

GTQ⟨m,n⟩ asserts m is greater than or equal to n.

Infix: m >= n.

LTQ⟨m,n⟩ asserts m is less than or equal to n.

Infix: m <= n.

C. Current Implementation

Although the design of the QA4 system is not complete, an initial implementation has been started to allow feedback and experimentation with many of the QA4 ideas. The implementation is being done in the LISP system on the PDP-10 and can be separated into three parts:

(1) Input Output. An expression parser to take the QA4 infix syntax into a prepolish or internal format has been written. The parser uses the BIP (Appendix E) package and has the advantage of being very easily modifiable. Similarly, an output function to take the internal expression form and output a corresponding infix version has been written. Finally, a top-level command language was designed to allow the user to enter and fetch expressions and properties of expressions from the data base and to do a variety of other simple tasks. This top-level function essentially interfaces the input output functions with the expression manipulation package.

(2) Expression Storage and Manipulation

The Discrimination Net. In order to allow arbitrary semantic properties to be assigned to expressions and to allow expressions to be retrieved by these properties, a

discrimination net has been implemented to store all QA4 expressions. Internally, a QA4 expression is a property list consisting of a property EXPV, whose value contains the syntactic information about the expression, and whose remaining properties are semantic. When an expression is stored in the net, a discrimination on the syntactic properties of the expression is made to determine whether or not the expression has already been stored in the net. If so, the old net expression is returned, and, if not, the new expression is added to the net. Thus the net contains only one copy of each expression stored in it. Moreover, a check for expression equivalence up to bound variable names is being added so that two QA4 expressions that are identical except for the names of their bound variables go into the same internal net expression. In order to store sets and bags in the net, an index is assigned to each element of a set or bag expression the first time it is stored. If the same set is then stored a second time (perhaps with some expressions permuted), the elements are first sorted by the index numbers and then discriminated upon syntactically. Thus if a user types in the set {A,B,C}, the elements are assigned indices A ← 1, B ← 2, C ← 3. If the set {C,B,A} is entered, it is sorted into {A,B,C} and then found to already occur. The net functions also maintain statistics concerning the number of references made to each expression and discrimination for future optimization.

Contexts. For the purpose of doing proofs by contradiction and conditional proofs (i.e., to prove A ⇒ B assign A true and prove B), a context scheme was required in which a given expression in the net has a value relative to a context. Thus, in the above example, if B is proved true, it is true only in

the context in which A was assigned true. In order to have this ability, each expression in the net has a value property that consists of a context name (LISP atom) and a corresponding value for that name. A context c is an ordered list (a_1, ... a_m) of such names. If the value of an expression in context c is desired, the list ($a_1,\ldots,a_n$) is examined in order until the first a_i is encountered for which the expression has a value--which is declared to be the value in that context. Thus to prove $A \Rightarrow B$ in a context $c = (a_1,\ldots,a_m)$ we can create a new context name a_o, assign A true in a_o, and prove B in the context $c^1 = (a_o,a_1,\ldots,a_n)$. If B is found to be true in c^1, then $A \Rightarrow B$ is assigned true in context c. The context mechanism is also useful for bound variable expressions: to perform some operation on the body of the expression in the context where the identifiers in the bound variable have been assigned certain values.

<u>Equality Partitions</u>. The efficient treatment of the equality predicate is crucial to the operation of any problem-solving system. Rather than axiomatize the equality rules, we have built them into the QA4 system by introducing equality partitions. Each expression in a context has (as its value property for that context name) the set of expressions known to be logically equal to it in that context. When two expressions are asserted or proved equal in a context, their "equality sets" are merged to form a new set for each. Moreover, each expression has (in context) a set of sets of expressions that are known to be unequal to the given expression. That is, each set in the "unequal set" contains a set of expressions known to be not all equal. Again, when a new equality assertion is made, these sets are correspondingly updated. Consequently, whenever

an equality assertion causes a contradiction via the equality rules, it is immediately known. An additional advantage to maintaining the equality information is to be able to select the "best" expression equal to a given expression for a certain purpose.

Primitive Functions. A variety of primitive QA4 set and bag functions have been implemented. In particular, the functions UNION, DIFFERENCE, INTERSECTION have been written for finite sets and bags, and these functions are used as the basis for the equality partition code.

Simplification. A simplification package has been written to simplify algebraic and logical expressions. The simplification function expands products, collects like terms, deletes zeros and zero factors, and so on.

3. Strategy Control. The strategy control primitives involve operations such as evaluating a set of expressions and returning when one of the evaluations succeeds, or when all the evaluations succeed, or when "progress" has been made on any one of the set. Ideally, one would like the evaluations to proceed in parallel, be able to communicate results to each other, and possibly remain in suspended animation for a period of time until invoked to proceed. To achieve this type of flow of control, a co-routine package has been implemented within the LISP system. Unlike the LISP flow of control, which must return from any started evaluation, the co-routines permit processes to be started, interrupted for the purpose of invoking any other process, and returned to by a completely different route. We plan to use the co-routine facility to monitor and control QA4 strategies and effort allocations.

Page Intentionally Left Blank

VI HARDWARE AND MAINTENANCE

The bulk of the hardware effort during this first period has been associated with converting the robot from the SDS 940 to the PDP-10 system. This has been divided into two major parts: rebuilding the TV A/D converter so as to interface it with the PDP-10 via the DF-10, which is a high-speed direct port to memory, and converting the robot interface to operate with the PDP-15 satellite computer.

The new 5-bit TV A/D converter has been designed, built, and is currently being debugged on line with the PDP-10. A digitized picture consists of 5-bit samples taken on the first of two interlaced fields with either 120X120 or 240X240 resolution. Each field is divided into 240 rows and 240 columns. In the low-resolution (120X120) mode the converter samples every other line and the odd numbered columns. In the high-resolution (240X240) mode the converter samples every line at the odd numbered columns, followed by a second pass sampling every line at the even numbered columns. External supervision of the converter will be provided by the PDP-15.

Conversion of the robot interface to operation with the PDP-15 is well advanced, with the primary obstacle to completion being the delay of arrival of the PDP-15.

The new fixed control unit (FCU) functions in both an off-line and an on-line mode. When the robot is on line the PDP-15 has direct control via the FCU, and the vehicle appears to it as a single peripheral device. The only signal that is recognized from the manual console is emergency stop. PDP-15 communication with the vehicle will consist of

(1) Direct program controlled transfers

(2) Data transfers via the PDP-15 I/O processor

(3) Robot interrupts via the automatic priority system.

As with the SDS 940, the basic unit of information transfer is the 8-bit character.

In the off-line mode the PDP-15 has access neither to the vehicle nor to the FCU. The console has exclusive control of the vehicle. This also means that local operation and checkout of the vehicle is possible without disturbing the PDP-15.

A PDP-15 simulator has been built to enable checkout of the converted FCU so as to minimize checkout time once the PDP-15 does arrive. Estimated date of completion for this effort, assuming present delivery schedules, is 15 June 1970.

REFERENCES

1. N. J. Nilsson et al., "Application of Intelligent Automata to Reconnaissance," Final Report, Contract AF 30(602)-4147, SRI Project 5953, Stanford Research Institute, Menlo Park, California (December 1968).

2. L. S. Coles et al., "Applications of Intelligent Automata to Reconnaissance," Final Report, Contract F30602-69-C-0056, SRI Project 7494, Stanford Research Institute, Menlo Park, California (November 1969).

3. R. O. Duda and P. E. Hart, "Experiments in Scene Analysis," SRI AI Group Technical Note 20, Stanford Research Institute, Menlo Park, California (January 1970).

4. C. R. Brice and C. L. Fennema, "Scene Analysis of Pictures Using Regions," SRI AI Group Technical Note 17, Stanford Research Institute, Menlo Park, California (November 1969).

5. C. R. Brice, C. L. Fennema, and S. A. Weyl, "AROS, Algorithms for Partitioning a Picture," SRI AI Group Technical Note 18, Stanford Research Institute, Menlo Park, California (January 1970).

6. J. A. Robinson, "Mechanizing Higher-Order Logic," in Machine Intelligence 4, Meltzer and Michie, eds. (Edinburgh University Press, Scotland, 1969).

Page Intentionally Left Blank

Appendix A

PDP-15 SIMULATOR

Page Intentionally Left Blank

April 1970

PDP-15 SIMULATOR

by

John K. Ellis
Leonard J. Chaitin

Artificial Intelligence Group
Technical Note 25

SRI Project 8259

This work was supported by the Advanced Research Projects Agency and the National Aeronautics and Space Administration under Contract NAS12-2221.

Page Intentionally Left Blank

ABSTRACT

This report describes briefly a PDP-15 simulator and its assembler, both of which were written on the PDP-10. The instruction repertoire for the simulator is complete except for input/output transfer instructions. The assembler provides an optional assembly and symbol table listing but currently has no pseudo-op or macro capabilities.

Page Intentionally Left Blank

OPERATING PROCEDURE

Transfer the binary file COD15.DAT from the DECtape to the disk using PIP:

```
.R PIP↵
*DSK:/B/X←DTAn:COD15.DAT↵
```

On the DECtape are core images of DDT loaded with three different assembler/simulator combinations:

SMLTR1.SAV	Assembles and loads PDP-15 code. SMLTR1.SAV optionally outputs binary code to a disk file called BIN15.DAT and can be used to obtain an optional assembly listing together with an optional alphabetized symbol table listing.
SMLTR2.SAV	Assembles and loads PDP-15 code. SMLTR2.SAV optionally outputs binary code to a disk file called BIN15.DAT but makes no provision for any kind of listing; however, SMLTR2.SAV does run in only 24K of memory and should be used if 78K of memory is not available.
SMLTR.SAV	Takes the place of a loading procedure and uses the binary file BIN15.DAT generated by either SMLTR1.SAV or SMLTR2.SAV.

If 78K of memory is available, use Procedure 1 described below; otherwise use Procedure 2, which requires only 24K of memory. In either case the program assumes that the code to be assembled is located on a disk file called PGM15.DAT. Procedure 1 or 2 should be used once for each different PGM15.DAT; Procedure 3 can then be followed as long as PGM15.DAT is not changed.

1. SMLTR1.SAV (78K)

The following sequence of commands turns control over to DDT:

```
.GET DTAn SMLTR1.SAV↵
JOB SETUP↵
↵
.DDT↵
.
```

The following notation is observed in console examples: computer typeouts are underlined, ↵ denotes a carriage return, $ designates the ALTMODE key, and ↑C means control C.

Insert the appropriate breakpoints (see section called Debugging Technique); then type $G. The assembler will print out the three messages shown below.

OUTPUT? TYPE Y OR N

If output to the disk file BIN15.DAT is desired, type a Y followed by a carriage return. A disk file called BIN15.DAT is created and used later in Procedure 3. If output is not desired, type an N followed by a carriage return.

LISTING? TYPE Y OR N

If an assembly listing is desired, type a Y followed by a carriage return. The assembler will output the following type of listing on the line printer:

Sequence No.	Memory Location	Octal Code	PDP-15 Code
10	100_8	707724	EBA;
20	101_8	111620	PASS1: JMS TTWRIT;
30	102_8	000204	CAL M9MSG;
⋮	⋮	⋮	⋮
52690	12534_8	000000	SRCBUF: BLOCK 44;
52700	12600_8	000000	PTPBUF: BLOCK 1000;
52710	13600_8	000000	END;

If an assembly listing is not desired, type an N followed by a carriage return.

INDEX? TYPE Y OR N

If an alphabetized symbol table is desired, type a Y followed by a carriage return. The assembler will output a symbol table listing on the line printer. If a symbol table is not desired, type an N followed by a carriage return.

2. SMLTR2.SAV (24K)

The following sequence of commands turns control over to DDT:

```
.GET DTAn SMLTR2.SAV⏎
JOB SETUP⏎
⏎
.DDT⏎
```

Insert the appropriate breakpoints (see Debugging Technique); then type $G. The assembler will print out the following message:

OUTPUT? TYPE Y OR N

If output to the disk file BIN15.DAT is desired, type a Y followed by a carriage return. A disk file called BIN15.DAT is created and used later in Procedure 3. If output is not desired, type an N followed by a carriage return.

3. SMLTR.SAV (18K)

The following sequence of commands turns control over to DDT:

```
.GET DTAn SMLTR.SAV↵
JOB SETUP↵
↵
.DDT↵
```

Insert the appropriate breakpoints (see Debugging Technique); then type $G.

DEBUGGING TECHNIQUE

Each time the simulator encounters a HLT instruction in a PDP-15 program, it prints out the following message on the teletype:

DEBUGGING? TYPE Y OR N

If you are trying to debug a PDP-15 program, type a Y followed by a carriage return. The PDP-15 instruction HLT will jump to memory location QNOP in the simulator where a breakpoint can be set using DDT. By varying the location of HLT instructions, DDT can then be used to assist in debugging a PDP-15 program. If you want to use this feature, set a breakpoint at memory location QNOP in the simulator, replace the given PDP-15 instruction by a HLT instruction, and replace the contents of memory location QLOC in the simulator by the given PDP-15 instruction. If, after a breakpoint halt occurs at memory location QNOP in the simulator, you want the position of the HLT instruction to stay the same, simply do an $P. However, if you want to change the position of the HLT instruction, set a breakpoint at memory location GETNXT+2 in the simulator. Do an $P, and when a breakpoint halt occurs at memory location GETNXT+2 in the simulator, replace the HLT instruction with the contents of memory location QLOC in the simulator, remove the breakpoint at memory location GETNXT+2 in the simulator, and proceed as above.

If you are not trying to debug a program, type an N followed by a carriage return. The PDP-15 instruction HLT will jump to memory location COMMON in the simulator, where the contents of the three timers are printed out, thus:

```
PDP15 TIME 5466.40 MICROSECONDS↵
IO WAIT TIME 2886 MILLISECONDS↵
RUNTIME 0 MIN, 0.50 SEC↵
↵
EXIT↵
↑C
↵
.
```

PDP15 TIME shows the total execute time required by the PDP-15 program in microseconds. Since the automatic priority interrupt system is not currently simulated, input/output is accomplished using program-controlled transfers. The input instructions KRB,KSF and the output instructions TSF,TLS are currently programmed to be used together in the following manner only:

```
TSF;          KSF;
JMP .-1;      JMP .-1;
TLS;          KRB;
```

IO WAIT TIME, therefore, shows the total teletype input/output wait time required by the PDP-15 program in milliseconds. RUNTIME shows the actual time required by the simulator to run the PDP-15 program.

SIMULATOR ERROR MESSAGES

If an illegal instruction or an illegal memory reference is encountered in a PDP-15 program, the simulator types out, respectively, the following two error messages:

ILLEGAL INSTRUCTION AT LOCATION 137↵

ILLEGAL MEMORY REFERENCE AT LOCATION 137↵

together with the contents of the three timers (see above). Since the simulated program counter is decremented in either case, the location given corresponds to the actual PDP-15 instruction which caused the error message to be printed out on the teletype.

SIMULATOR DESCRIPTION

The size of the simulated machine can be varied from 1K to 128K of memory by adjusting memory location SIZE.

The assembler generates code which the simulator assumes to be in the following format:

```
PDP15:        data,,garbage  \
PDP15+1_8:    data,,garbage   |
              :                > Reserved addresses
PDP15+77_8:   data,,garbage  /
PDP15+100_8:  data,,garbage  \
PDP15+101_8:  data,,garbage   |
              :                > PDP-15 program
PDP15+   _8:  data,,garbage  /
```

The PDP-15 program is loaded beginning with memory location PDP15+100_8 (i.e. location 100_8 in the simulated machine).

The right half of each data word is then zeroed so that the code is in the following format:

```
PDP15:          data,,0  \
PDP15+1₈:       data,,0   |
   :                      > Reserved addresses
PDP15+77₈:      data,,0  /
PDP15+100₈:     data,,0  \
PDP15+101₈:     data,,0   |
   :                      > PDP-15 program
PDP15+    ₈:    data,,0  /
```

Unless the normal program sequence is altered, program control is determined by the following two instructions:

```
GETNXT: MOVE T,PDP15(PC)
        JSR INCPC
```

The program counter (PC) is originally set to 100_8. INCPC is a subroutine that increments the program counter modulo 4096 (modulo 8192 if bank mode addressing is in effect).

The simulator decodes PDP-15 instructions by scanning in the order shown below until one of two things happens: (1) the instruction is discovered to be a legal, currently implemented PDP-15 instruction, in which case the program jumps to the appropriate subroutine, or (2) the instruction is discovered to be an illegal or currently unimplemented PDP-15 instruction, in which case the program jumps to an illegal instruction subroutine.

The program checks the contents of accumulator T for one of six input/output instructions now implemented: TSF, TLS, KSF, KRB, EBA, or DBA. If T doesn't contain an input/output instruction, the program jumps on the operation code (bits 0-3):

```
ENTRY:  ROT T,4
        JRST @.+1(T)
```

to a subroutine corresponding to a memory reference instruction or to one of three general instruction groups: index operate/input/output instructions (IO), EAE instructions (EAE), or microcoded instructions (MCRCD).

1. Memory Reference Instructions

All memory reference instructions JSR to the subroutine MODE. If bank mode addressing is in effect (memory location BNKFLG is set to all 1's) the program JRST's to the subroutine BNKMOD. If bank mode addressing is not in effect (memory location BNKFLG is set to 0) the program jumps on the address mode:

```
MODE:   0
        HRRI T,0
        SKIPE BNKFLG
        JRST BNKMOD
        ROT T,2
        JRST @.+1(T)
```

to subroutines which handle one of four address mode combinations: direct (NONE), indexed (BIT5ON), indirect (BIT4ON), or indirect-indexed (BOTH). Any one of these combinations can be handled by the following three subroutines: current page address (CPADR), indexing (INDEX), and indirect addressing (INDRCT). INDEX and INDRCT both JSR to the subroutine MEMORY, which JRST's to an illegal memory reference subroutine if the memory capacity (depending upon memory location SIZE) is exceeded.

2. Index Operate/Input/Output Instructions (IO)

If bank mode addressing is in effect, the program JRST's to an illegal instruction subroutine, since all indexing operations are eliminated in favor of bank addressing. If bank mode addressing is not in effect, the program tests for input/output instructions; any input/output instruction discovered at this point is not currently implemented and causes a JRST to an illegal instruction subroutine.

The program then jumps on bits 5-8 to the appropriate subroutines:

```
IO:     SKIPN BNKFLG
        TLZN T,400000
        JRST ILLGLI
        HRRI T,0
        ROT T,5
        JRST @.+1(T)
```

3. EAE Instructions (EAE)

Figure 7-8 and Table 7-2 in the PDP-9 User Handbook illustrate EAE instruction microcoding. The EAE instructions are microcoded using only logical test instructions:

```
EAE:    TLZE T,400000          SHAL:   ADDI TIMER,442
        JRST SHAL                           :
XCLQ:   TLZE T,200000                       :
                                       JRST XCLQ
```

together with a switch which jumps on bits 9-11:

```
BRNCH:  HRRI T,0
        ROT T,10
        JRST @.+1(T)
```

4. Microcoded Instructions (MCRCD)

Figure 7-9 in the PDP-9 User Handbook illustrates the microcoded instructions; however, the actual scanning sequence appears to be (from left to right):

0=ORof 1=ANDof	SNL SZA SMA SZL SNA SPA	CLA	CLL	OAS	CML	CMA	Bit7=0 Bit7=1	RAR RAL RTR RTL	HLT
8	9 10 11	5	6	15	16	17	7	13 14	12

which is undocumented but consistent with all available PDP-9/PDP-15 documentation.

The microcoded instructions are microcoded using only logical test instructions in a manner analogous to the EAE instructions. The microcoded instruction OAS (OR Console Accumulator Switches to the Accumulator) is not currently implemented and generates an illegal instruction.

USING THE ASSEMBLER

The assembler (until we get MACRO-15 working) effects to find a disk file called PGM15.DAT. For editing purposes this file should have sequential line numbers incremented by 10, thus:

```
10      ABC: LAC .+2;
20       DAC ABC#;
30       HLT;
40      DEF: DATA 777;
```

The assembler will take one instruction per line. Each instruction must end with a semicolon. The format is free field--all spaces are ignored. There are three fields per instruction: the label field (optional), the op-code field, and the variable field.

The label field (if present) is identified by a colon following the label. No more than six characters are permitted; the first character must be a letter. Only letters and numbers are permitted.

The op-code field contains any legal PDP-15 code (as defined in the manual) plus the pseudo-ops DATA, BLOCK, and END. The DATA instruction generates a word of numeric data. The BLOCK instruction reserves a block of zeroed words. The size of the block is designated in the variable field. The END instruction must be the last instruction in a program and must be present.

The variable field may contain a label, a period, or a number, optionally followed by a + or - followed by another number. All numbers are taken to be octal. If a decimal number is wanted, it must be followed by a $. The period stands for the current setting of the location counter, which is initialized to 100_8. The variable field may be preceded by an *, which indicates indirect addressing, and may be followed by a #, which indicates indexing.

Appendix B

A LISP-FORTRAN-MACRO INTERFACE FOR THE PDP-10 COMPUTER

Page Intentionally Left Blank

November 1969

A LISP-FORTRAN-MACRO INTERFACE FOR THE PDP-10 COMPUTER

By:

John H. Munson

Artificial Intelligence Group

Technical Note No. 16

SRI Project No. 8259

(Supersedes Technical Note No. 12)

Page Intentionally Left Blank

SUMMARY

An interface has been devised for use on the PDP-10 computer that allows FORTRAN (or FORTRAN-compatible MACRO) subroutines and functions to be run under the LISP operating system (specifically, the LISP written at Stanford University, and described in Stanford Artificial Intelligence Project Note SAILON 28, by Lynn Quam). A considerable effort has been made to endow the interface with generality and ease of use, as much as could be achieved without tampering with the FORTRAN and LISP operating systems and compilers. The operating features of the interface are as follows:

(1) The interface and the various FORTRAN subprograms are loaded with the regular loader for relocatable programs that is available under the LISP system.

(2) FORTRAN subprograms may be called at will from within LISP. To the LISP programmer, these subprograms look exactly like LISP functions. Up to four arguments may be passed. The arguments and the returned value may be integers, floating-point numbers, LISP atoms, or other S-expressions.

(3) An optional call-by-name mechanism is incorporated: after the execution of the FORTRAN subprogram, LISP may access the calling arguments, which may have been changed by the subprogram.

(4) At any point or points in the FORTRAN subprogram structure, a function call may be made back into LISP. Again, the arguments may be of various types. The recursion ends here, however. Neither the interface nor the FORTRAN system, as presently constituted, can handle a second recursive entrance into FORTRAN.

The current form of the interface represents two major improvements over the earlier version, namely, the handling of non-numeric arguments and the ability to call back into LISP. By thus allowing FORTRAN to "dip into" LISP conveniently, we reap at least three benefits. The first is simply a vast improvement in intercommunication, with FORTRAN now able to access information within LISP data structures. The second is the ability of FORTRAN to invoke LISP functions for those activities (notably input/output) that are presently missing because the FORTRAN operating system

is not in operation. (We may be driven, however, to remedy this situation more directly in the future.) The third benefit is the use of the improved interface as a link for two-way communication in which large structures or quantities of data are explicitly shared by the LISP and FORTRAN subsystems, for example in the form of EXARRAY's.

TERMINOLOGY

The term "FORTRAN" is used herein as a shorthand for any and all program codes that can be loaded by the PDP-10 relocatable program loader operating under the LISP system. In general, these will include subprograms written in MACRO as well as in FORTRAN. Top-level subprograms called directly through the interface, and those called from FORTRAN subprograms, must be FORTRAN-compatible in their calling sequences. The same applies to subprograms calling back to LISP through the interface. Elsewhere within the "FORTRAN" side, FORTRAN compatibility need not be maintained, as long as consistency is maintained at each point.

Throughout this paper, we use the term "subprogram" to refer generically to SUBROUTINES and FUNCTIONS in FORTRAN. FUNCTIONS and SUBROUTINES can be used interchangeably if the returned value, or lack thereof, is of no consequence. At present (contrary to the manuals) FORTRAN FUNCTIONS do store back into their calling arguments and thus allow the call-by-name mechanism. Furthermore, listings indicate that SUBROUTINES may not save and restore all arguments. Thus, it is perhaps preferable to write everything as functions.

ARGUMENT TYPES AND CONVERSIONS

Both FORTRAN and LISP recognize two types of numeric quantities, integers and floating-point (or real) numbers. Since the FORTRAN and LISP systems represent each of these differently, the interface performs a conversion on every numeric argument (LISP functions NUMVAL and MAKNUM

are invoked to do these). Pointers to non-numeric atoms and S-expressions in LISP are transmitted unchanged to FORTRAN, where they appear as single-word variables, with the pointer in the right half word and zero in the left half.

FORTRAN has a limited data type called "literal," used for symbolic information. Literal constants such as 'ABC123' may be coded into a FORTRAN program and are stored as consecutive words of 7-bit ASCII characters, five to a word, left justified, with the last word filled out with blanks if necessary, and followed by a word of all zeroes. Since there is no literal variable type, any _variable_ literal must be made up within a variable or array of some other type. _The interface will convert a FORTRAN literal into the LISP pointer to the atom whose print name matches the literal_. (The interface scans the literal, character by character, until it encounters either an ASCII blank or zero. The maximum length allowed is 25 characters.) How the interface is aware of a literal is described below.

Table I summarizes the conversions that are applied by the interface to every argument (when LISP calls FORTRAN or vice versa) and to every result value when the corresponding returns are made. Also, when a call-by-name argument reference is made by LISP, the argument is converted from a FORTRAN data type back to a LISP data type.

TABLE I

CONVLF and CONVFL Conversions

LISP Quantity	Conversion	FORTRAN Quantity Type	FORTRAN Quantity Code
Integer (INUM or FIXNUM)	↔	Integer*	0
Real (FLONUM)	↔	Real*	2
Pointer to a non-numeric S-Expr. (machine address)	←	Logical*	3
Pointer to a non-numeric S-Expr. (machine address); Pointer to Atom (machine address)	→	Octal	4
Pointer to Atom (machine address)	←	Literal	5
Pointer to Atom (machine address)	←	Dbl. Prec.*	6
Pointer to Atom (machine address)	←	Complex*	7

——→ Indicates LISP-to-FORTRAN conversion (CONVLF)

←—— Indicates FORTRAN-to-LISP conversion (CONVFL)

* Indicates existence as FORTRAN variable type

The LISP-to-FORTRAN conversion routine in the interface, CONVLF, determines the nature of each LISP quantity by examining it (with NUMBERP). If numeric, the quantity is converted with NUMVAL; if not, it is passed unchanged. The FORTRAN type-code is also generated and stored (even though called FORTRAN routines seem to ignore the codes), so that in the case of a call-by-name argument the reverse conversion can be performed properly later. The "octal" code is generated for all non-numeric pointers, because intuitively it seems most appropriate.

The reverse, or FORTRAN-to-LISP conversion routine (CONVFL), determines the type of the FORTRAN quantity by examining the type code as shown in Table I. These type codes are taken from the standard FORTRAN subprogram calling sequence (see the PDP-10 FORTRAN IV manual, Appendix 4). If a FORTRAN routine is to call LISP with an "octal" or "literal" variable quantity, that quantity must be stored in (or EQUIVALENCE'd to) a logical variable (for an octal quantity) or a complex or double-precision variable

(for a literal quantity) so that the latter may be used in the calling sequence to establish the type for the conversion. This dodge requires a slight amount of care, but it should not impose any real hardship. On the other hand, an octal or literal constant may be coded into the calling sequence without further ado.

CALLING FORTRAN FROM LISP

A call to a FORTRAN subprogram appears in LISP exactly as would a call to a LISP function of the same name; thus, the FORTRAN function

```
FUNCTION ITEST (I,J,X,Y)
LOGICAL Y
etc.
```

might be called from LISP as

```
(ITEST 3 JJ 4.0 NIL)   .
```

To prepare LISP for this, we define a dummy function ITEST in LISP as follows:

```
(DE ITEST (I J X Y) (IFORT4(FORTREF(QUOTE ITEST))I J X Y)).
```

This causes ITEST as just defined to occur as a real LISP function, so it can be called in LISP. When it is called, FORTREF takes the name "ITEST" to the loader symbol table, and returns with the address of the FORTRAN function ITEST (which is then stored on the property list of ITEST under "FORTFUNC"). IFORT4 is an entry into the interface itself. The interface takes the function address (which it receives as its first actual argument, in accumulator 1) and prepares a FORTRAN calling sequence to that address. The interface then performs the CONVLF conversion on the remaining arguments, stores them in the calling sequence, and enters the FORTRAN subprogram.

When the FORTRAN subprogram does a RETURN, control comes back to the interface. The value of the function (or garbage in the case of a subroutine) is converted back by CONVFL, and control returns to LISP.

For each FORTRAN subprogram to be thus called from LISP the only necessary preliminary is a dummy function definition similar to that above. The identifier IFORT4 is varied in two ways, as appropriate to the subprogram called. The terminal digit tells the number of arguments (limited to 4 by a constraint of LISP), and the initial letter is I for an integer result to be returned, null for a floating-point (real) result, and P for a pointer result. Thus, the interface provides the following entries:

Number of Arguments	Integer Result	Real Result	Pointer Result
None	IFØRTO	FØRTO	PFØRTO
1	IFØRT1	FØRT1	PFØRT1
2	IFØRT2	FØRT2	PFØRT2
3	IFØRT3	FØRT3	PFØRT3
4	IFØRT4	FØRT4	PFØRT4

(Pardon our previous laxness about slashed O's.)

Upon return to LISP, accumulators 0 and 6-17 are restored to the values they had when LISP called the interface. Accumulator 1 holds the result.

As in most FORTRAN programming, it is the programmer's responsibility to ensure that the types of the arguments and the result are agreed upon between the called program and the calling program, given the transformation performed by the interface.

THE CALL-BY-NAME MECHANISM

A distinction is made in FORTRAN between call-by-name and call-by-value. In a call by name, the called subprogram is given access to the

location of the argument in question where it resides within the calling program; hence, if the called program changes the value of the argument, it changes its value within the calling program as well. In a call by value, on the other hand, the called subprogram is given access only to a copy of the value of the argument and cannot change its value within the calling program. PDP-10 FORTRAN SUBROUTINES and (contrary to the manual) FUNCTIONS operate on a call-by-name basis at present.

An analogous distinction may be made in LISP. LISP usually operates on a call-by-value basis, in that if a function (FOO ATOM) is executed, FOO receives the value of ATOM and cannot change the fact that ATOM is bound to that value. Only if (FOO (QUOTE ATOM)) is executed is FOO able to get at ATOM and change its binding.

The interface allows the LISP programmer the option of accessing the arguments of a called FORTRAN subprogram after the subprogram has been executed, thus creating the effect of a call by name. To do this, one of the following functions is executed in LISP:

(ARG1) (ARG2) (ARG3) (ARG4) ,

corresponding to the first through fourth arguments of the earlier calling sequence. Each function returns the current value of the argument, as it was left by the most recently called FORTRAN subprogram. (The interface properly converts the returned argument back to the LISP type that it had when the subprogram call was made.)

For example, if the called FORTRAN subprogram

FUNCTION ITEST (I,J,X,Y)

executed the statement

J = J + 1 ,

as if it were called from LISP as

(ITEST 3 JJ 4.0 NIL)

where JJ had been SETQ'ed to 5, this would be a call by value, and the value of JJ in LISP would be unchanged. But the form

. . . (ITEST 3 JJ 4.0 NIL)(SETQ JJ (ARG2)) . . .

would change the value of JJ in LISP to 6.

The call-by-name mechanism affords a convenient means for passing back to LISP information in addition to the function value, without bothering to create EXARRAY's or other mechanisms.

CALLING LISP FROM FORTRAN

At any point within the FORTRAN subprogram structure, a function call may be made back into LISP by invoking the following "FORTRAN function":

LISP(lisp-fn, arg1,arg2, . . . ,argn) .

This behaves as a regular function in FORTRAN, i.e., it can be coded into an arithmetic assignment statement and it returns a value.

The argument "lisp-fn" must convert, under CONVFL conversion, either to a pointer to the atom LISPFN or to a pointer to the executable compiled code for LISPFN, where LISPFN is the name of the function to be evaluated in LISP. Therefore, "lisp-fn" in FORTRAN must be either the pointer itself (in which case it must be given a logical or octal type) or the FORTRAN literal 'LISPFN' (in which case it must be given a literal, double precision, or complex type). In normal usage, of course, where it is merely desired to call a specific LISP function from a specific point in the FORTRAN structure, it suffices to code the literal constant (of up to 25 characters) directly into the calling sequence:

LISP('LISPFN',arg1,. . .).

The number of arguments for the lisp function, n, may range from zero through five. (The interface automatically determines the length

of the calling sequence, so only the single entry point "LISP" is needed.) The interface performs CONVFL conversion on the arguments, with their types as specified in the FORTRAN calling sequence. On return from LISP, the interface examines the result to determine its LISP type and performs CONVLF conversion on it.

The values of all of FORTRAN's accumulators are saved by the interface and restored to FORTRAN before returning. Whenever LISP is entered (either by a call or a return from FORTRAN), the values of LISP's accumulators are restored to those that were saved at the last exit from LISP. Thus, both LISP and FORTRAN see the necessary continuity of storage in their accumulators.

MISCELLANY

The interface, the FORTRAN functions, and DDT if desired, are loaded under LISP function (LOAD) by naming the files in which they reside. When the loader is terminated with the escape key, any called-for FORTRAN library functions are loaded. The status of the FORTRAN operating-system routines is unclear; in any case, they are not operative in the current arrangement because their UUO's conflict with those of LISP.

Other preliminaries amount to making all the xFORTn and ARGn entries of the interface available in LISP (through GETSYM), making certain LISP functions available to the interface (through PUTSYM), and establishing the dummy LISP functions and FORTREF.

The programmer must be aware that when control passes to LISP, list structures are subject to garbage collection under the rules of LISP. Thus, any pointers into LISP that are stored in FORTRAN across a time when LISP is entered must refer to structures that are protected from garbage collection.

The interface uses the LISP UUO "CALL" (see SAILON 28, Appendix 3) to enter a called LISP function. CALL accepts a pointer to an atom that has a SUBR or EXPR function definition; otherwise, it assumes the pointer is to compiled code. This means that <u>other functional forms, such as</u>

MACRO's, cannot be called directly this way. To get at these, one must shield the MACRO with a dummy function definition, or go through EVAL, or fetch pointers to the appropriate codes and use them directly, or perform some other trick. The result of calling a MACRO directly, or calling a named function that is not established in LISP, is generally to enter the atom header in LISP or in the interface and cause a crash.

Listings of the interface and other routines involved are available from the author.

Appendix C

SOME REMARKS ON RESOLUTION STRATEGIES

Page Intentionally Left Blank

April 1970

SOME REMARKS ON RESOLUTION STRATEGIES

by

Robert E. Kling

Artificial Intelligence Group

Technical Note 28

SRI Project 8259

The research reported here was supported by the Advanced Research Projects Agency and the National Aeronautics and Space Administration under Contract NAS12-2221.

Page Intentionally Left Blank

SOME REMARKS ON RESOLUTION STRATEGIES

The following comments are exerpted from a letter I wrote discussing some problem-dependent aspects of resolution-logic theorem proving. The underlying focus of these remarks is the nature of the information that a user needs to specify for a problem-oriented strategy to be employed by a multi-strategy system. However, I was more concerned with summarizing some of my experiences and with articulating certain questions than with reaching particular conclusions about the necessary ingredients for such a language, let alone deriving a preliminary language design. Nevertheless, some of these comments seem to have been interesting to other readers and may benefit from a wider circulation.

I'd like to classify the various strategies I know, some of which we currently use on QA 3.5 under three headings, and then describe a few additional details of strategic nature which aren't subsumed in this classification.

Attention (Ordering) Strategies:

(Which clause pair shall we consider next?)

(1) Unit Preference (WOS) - QA3 - A classical strategy.

(2) Preference Set (Kling) - QA3 - Partition memory, allowing some axioms likely to be used in a given proof or "preferred status." Draw in other axioms only if the preferred axioms don't lead to proof (by a preset level, number of search nodes, or other criterion). This is independent of unit preference.

(3) Splitting (Slagle) - Similar to backwards chaining. It entails developing an AND/OR subgoal tree in which goals are satisfied by resolution deductions through the tree to □. (Might be compatible with modified unit preference at each level of tree search, but it's really motivated by radically different considerations.)

Selection Strategies:

(Given a set of clauses, shall they be resolved?)

(1) T-Support (Wos) - QA3 - Classical.

(2) Hillclimbing (Green) - QA3 - Need a metric for the space.

(3) Predicate Filter (Kling) - QA3 - Accept a clause iff all its predicates are on a "filter list." (A crude way to trim the data base by "subject area" and relations likely to enter into a given proof--very useful with analogies.)

(4) Indicator Test (Kling) - QA3 - A helpful strategy for neglecting irrelevant but pregnant clauses in second-order domains like algebra. Some predicates--e.g., group[g;*] or map[f;x;y]--have "indicators"--e.g. the group operator *, the map function f, etc. Some clauses contain these "second-order predicates" and fully first-order predicates--e.g.

$$\text{subgroup}[h;g;*] \vee \text{subset}[h;g] \quad .$$

These clauses will not be resolved on the set predicates (in, subset, factorset, intersection, etc.) until all the indicators in the clause are fully instantiated.

(5) Ancestry Filter (Luckham) - QA3 - See his paper from Stanford AI Project.

(6) Merging (Andrews) - QA3 - See his paper in JACM.

(7) Length Plus Level Bound - QA3 - Set in advance.

(8) Term Depth - QA3 - Set in advance.

<u>Deletion Strategies</u>:

(Given a resolvent, should it be added to the active search tree?)

(1) Doublerestest (Kling) - Do not accept a resolvent unless it resolves with at least one node in the developed search space. (Actually more elaborate, but an incomplete strategy. Resolvents that fail this criterion must be saved and tried again, or used themselves if no double resolvents exist.)

(2) Subsumption (Robinson) - QA3.

(3) Forbidden States (Kling) - QA3 - Some values of state variables are "forbidden" on semantic grounds, and clauses containing states with these values are edited out.

(4) Answer-Units Only (Green) - QA3 - Keep clauses which only have an answer-clause which is a unit.

<u>Other</u>:

(1) Predicate Evaluation (Green) - QA3 - Associate a LISP form with a predicate so that the literal may be evaluated to T, NIL, or even return a different literal.

(2) Function Evaluation (Green) - QA3 - Associate certain functions with LISP forms, for evaluation.

I want to provide a simple example about the strategic use of axioms and how we simplify some of our searches by clever tricks. Since the example I develop uses predicate and function evaluations, I'll develop an example of predicate evaluation first.

Predicate evaluation has turned out to be an effective way to embed semantic information in a comparatively efficient way. Cordell's paper "Application of Theorem-Proving Techniques to Problem Solving" describes our state-transformation approach to problem solving. Consider a simple robot situation in which we want the robot to reach a box which is situated on a platform. To reach the platform top, the robot must roll up a wedge. (This is a loose analogue of the monkey-bananas problem.) A simple axiomatization may include the following axioms:

A1. $\forall (r\ w\ p\ \ell\ s)\ on[r;\ell;s] \wedge on[w;\ell;s] \wedge on[p;\ell;s]$
$\wedge\ at[w;p;s] \wedge at[r;p;s]$
$\rightarrow on[r;p;rollup[r;w;s]]$

A2. on[robot;floor;initial-state]

A3. on[$platform_1$;floor;initial-state]

A4. on[$wedge_1$;floor;initial-state]

A5. on[box_1;$platform_1$;initial-state] .

Axiom A1 states that if the robot, wedge, and platform are all on level ℓ and together in state s, then the robot can get onto the platform by rolling up the wedge. Now other action axioms are needed to develop a state s, such that at[robot;$platform_1$;s] and at[$wedge_1$;$platform_1$] are true. But A3 or A4 could clash with a partially instantiated derivative of A1 to yield

on[robot;$wedge_1$;rollup[r;w;s]]

or

on[robot;p;rollup[r;$platform_1$;s]] ,

etc.

One way of eliminating semantically senseless clauses like these is to use predicate evaluation for embedding types. For example:

A1′. $\forall (r\ w\ p\ \ell\ s)\ on[r;\ell;s] \wedge on[w;\ell;s] \wedge on[p;\ell;s]$
$\wedge\ at[w;p;s] \wedge at[r;p;s] \wedge wedge[w] \wedge platform[p]$
$\rightarrow on[r;p;rollup[r;w;s]]$.

If wedge[x] and platform[x] are associated with evaluable LISP forms that are T or NIL for appropriate (ground) arguments, then semantically senseless clauses will evaluate to T and be deleted. Semantically sensible clauses will be acceptable and the additional type literals will "disappear" after evaluation (to NIL).

Function evaluation is also helpful in embedding some semantics into our axiomatizations. We have a function p[x] which will unify (thus resolve) or subsume with any permutation of the argument list x. Thus in geometry, we will say triangle[p(A B C)] instead of triangle[A;B;C] and

$$\forall (x\ y\ z)\ \text{triangle}[x;y;z] \rightarrow \text{triangle}[z;x;y] \wedge \text{triangle}[y;z;x] \quad .$$

Consider the following theorem: The intersection of two Abelian groups is Abelian--i.e.

$$\text{int}[C;A;B] \wedge \text{abelian}[A;*_1] \wedge \text{abelian}[B;*_1] \rightarrow \text{abelian}[C;*_1] \quad .$$

In proving this theorem, if one knows that the intersection of two groups is a group, one should easily derive group[C;$*_1$]. So in the course of proving this theorem the following clauses may appear in the search space:

A1. $\sim$group[z;*] $\vee$ $\sim$int[z;x;y] $\vee$ $\sim$group[x;*] $\vee$ $\sim$group[y;*]

C1. group[A;$*_1$]

C2. group[B;$*_2$] .

Four resolvents may be derived:

R1. group[z;$*_1$] $\vee$ $\sim$int[z;A;A]

R2. group[z;$*_1$] $\vee$ $\sim$int[z;B;B]

R3. group[z;$*_1$] $\vee$ $\sim$int[z;A;B]

R4. group[z;$*_1$] $\vee$ $\sim$int[z;B;A] .

We really want <u>either</u> R3 or R4, and R1 and R2 are genuinely spurious. If we associate an evaluable form intf[x;y;z] with intersection[x;y;z], then we can easily throw away a clause that contains a term of the form int[z;x;x]. By using the function p[x], described above, in a new axiomatization, R3 and R4 may be compressed into a single clause z which would be like

R5. group[z;$*_1$] $\vee$ $\sim$int[z;p[(A B)]] .

This joint use of evaluation procedures is merely a clever tactical device. The unsolved problems that we face here include various ways of specifying the use of a particular axiom or lemma. Often a unit lemma is used once, to resolve with a particular axiom, and is then forgotten. Often an axiom, such as the preceding relationship between a pair of groups and their intersection, is quickly resolved to some simple form--e.g. R5. Or R5 is resolved

with the premise int[C;p[(A B)]] and is forgotten. Rarely are intermediate results like R5, or even $\text{group}[z;*_1] \vee \sim\text{int}[z;p[(A\ B)]] \vee \sim\text{group}[y;*_1]$ used on more than one line of reasoning. But we aren't sure how to specify the use of an axiom (or sequence of axioms) to focus on their key results (maximal resolvents) in a way we can specify as a problem-dependent user-supplied strategy.

Another problem-dependent strategy arises in our "three box" problem. Initially we have three boxes dispersed over a room, and the position of each box is noted--e.g., $\text{position}[A;p_1;\text{initial-state}]$ (Box A is initially at p_1). We then ask QA3 to find a sequence of actions that will bring the boxes together. The problem statement is

$$\exists(s_f\ p)\ \text{position}[A;p;s_f] \wedge \text{position}[B;p;s_f] \wedge \text{position}[C;p;s_f] \quad .$$

A wise problem solver would fix the initial positions of A, B, or C as the target position and move the other boxes. QA3, working with breadth-first search, sets up each of A, B, and C as a target position and problem solves in parallel. What kind of executive do we need to automatically restate the problem as

$$\exists(s_f)\ \text{position}[A;p_1;s_f] \wedge \text{position}[B;p_1;s_f] \wedge \text{position}[C;p_1;s_f]$$

or to shift the style of search on the initial problem?

Page Intentionally Left Blank

Appendix D

LISP TRACE PACKAGE FOR PDP-10

April 1970

LISP TRACE PACKAGE FOR PDP-10

by

Robert E. Kling

Artificial Intelligence Group

Technical Note 27

SRI Project 8259

Page Intentionally Left Blank

I've written a new LISP TRACE package which supercedes the current TRACE package. A fully descriptive memo will be available in a week or so; this is a brief introduction to facilitate earlier use.

* * * * * * *

1. All the functions described below are upwards compatible with any that are used in the system-associated TRACE package.

2. To access the package, type N when the LISP system asks whether you want TRACE?. Then type (INC SYS:␣BRKTRF) from within LISP. The new package uses about 3700 free words, compared to 1800 used by the original package. So allow 2K more free storage (or expect 2K less free space available).

3a. All the printing done by the system is effected by a global variable, PRINTLEVEL, which is initially set to 10.

3b. When the TRACE package encounters an error or if a function is broken, the system calls help[m] which prints a message m and then enters a READ-EVAL-PRINTN loop. printn[s], the print function, is responsive to PRINTLEVEL.

4a. To trace $fn_1 \ldots fn_k$, execute (TRACE FN1 FN2 ... FNK) as usual. All the arguments will be printed. Consider a function fn_2[x;y;z]. Suppose you only want to see the values of x and cdr(y); and you want only to see the length of the value of fn_2[x;y;z]. Then execute

(TRACE (FN2 X (CDR Y) ; ␣: (LENGTH FN2))) .

4b. If a function FN3 to be traced is compiled, the TRACE routine will respond with

(FN3 IS A COMPILED FUNCTION:␣ARGS = ?) .

Enter a list of function arguments. Thus, to trace subst[x;y;z], type (X Y Z) when asked for an argument list.

4c. To untrace $fn_1 \ldots fn_k$ execute (UNTRACE FN1 FN2 ... FNK). When a function is traced its name is added to the global variable ALLTR. (UNTRACE ALLTR) will untrace everything.

4d. The functions which are called by the TRACE package in tracing a function may not themselves be traced. These functions include: MAPC, MAPCAR, LENGTH, SUB1, ADD1, PLUS, *DIF, SUBST, CAR, CDR, GET, GETL, and CADADR.

4e. The tracing functions are heavily error protected with errset[s]. When an error occurs during tracing the message (HELP CONTROL) is printed and you have access to EVAL with variable bindings as saved local to the error location. In order to uplevel to the bindings as they are stored at the time the last traced function was entered, type (ERR). Successive evaluations of (ERR) will unwind you step-by-step through each level of traced-function calls. A control G will bring you to the top-level eval. After exiting from an error-interrupted trace, execute (RESET) to reinitialize variable bindings and restore certain global tracing parameters. Warning! Do not evaluate (RESET) within the TRACE package, but hit control G and exit to the top level first.

5a. To trace fn only when it is called by $fn_1 \ldots fn_k$, execute

(TRACEIN FN FN1 ... FNK) .

For example, to trace memq[x;ℓ] only when it is called by testfn1[ ] and testfn2[ ] execute

(TRACEIN MEMQ TESTFN1 TESTFN2) .

If you want to see only x and length (ℓ), execute

(TRACEIN(MEMQ X (LENGTH L); :) TESTFN1 TESTFN2) .

5b. To deactivate the tracein feature, for MEMQ, execute

(UNTRACEIN MEMQ) .

To still trace memq[x,ℓ] within testfns[ ], execute

(UNTRACEIN (MEMQ TESTFN1)) .

6a. If you only want to see a function's trace print when some predicate p is satisfied, then execute trshow[fn; p] for each function-predicate pair. For memq[x;ℓ] above,

(TRSHOW MEMQ(LESSP(LENGTH L) 10))

will only show a trace of MEMQ if length [ℓ] < 10.

6b. trunshow[fn_1;fn_2;fn_3...] reverses the effects to trshow [fn; p] for fn_1, fn_2, fn_3 ...

7a. I've written a simple break feature that stops the system when a specified function is entered and calls the HELP program. Executing break[fn; p] will halt fn upon entering if p is true. A message (FN BROKEN) is printed and the user has access to EVAL with the PRINTN feature. A broken function is halted just after its arguments are bound to the LAMBDA variables and are evaluated. To exit from a break, type OK.

7b. Unbreak[fn_1;fn_2;...] will unbreak the listed functions.

8a. tracet[ℓ] has been modified in several ways:

(1) The output format for SET-SETQ tracing is of the form: x ← 3.

(2) GET, GETL, REMPROP, and PUTPROP are traced along with SET and SETQ. Get[nam; prop] prints out as: PROP(NAM) = VALUE. If either <u>prop</u> or <u>nam</u> are on the list ALLSET the preceding printout will occur.

(3) The tracing may be turned off without having to reinsert a list of atoms to be traced.

8b. (TRACET) will start the SET-SETQ-GET... tracing.

(TRACET A1 A2 A3 ...) will trace each of A1, A2, A3...

(TRACET T) will turn the tracing on if it has been turned off.

(UNTRACET) destroys the tracing list ALLSET <u>and</u> turns off the tracing system.

(UNTRACET A1 A2 ...) untraces A1, A2 ...

(UNTRACET T) suspends tracing printouts but does not destroy the reference list.

9. I've modified <u>edit</u> to work with traced functions.

10. All these features are mutually compatible. No doubt hidden bugs are still lurking within the code. I'd appreciate a printout associated with any errors. Or, more effectively, SAVE your system at the time a peculiar error occurs and I'll be able to debug it quickly.

11. I'd appreciate any comments or suggestions regarding the ease or difficulty of using this system.

Appendix E

A LISP IMPLEMENTATION OF BIP

Page Intentionally Left Blank

February 1970

A LISP IMPLEMENTATION OF BIP

by

Richard E. Fikes

Artificial Intelligence Group

Technical Note 22

Page Intentionally Left Blank

INTRODUCTION

This document describes a LISP implementation of BIP (Basic Interface Package) on the PDP-10 computer. BIP is a set of programs designed by Allen Newell at Carnegie-Mellon University which provides the builder of large programming systems a capability for easily defining notational conventions to be used for interacting with a system.* The central routine in BIP is a translator which provides a symbol table and precedence-parsing facility. The entire package provides the following capabilities:

(1) Segmentation of an input stream of characters into "words"

(2) Association of a word to a particular internal symbol

(3) Recognition that some program (action) should be executed upon encountering a particular word in the input

(4) Retention of several symbols and their order of appearance as a context for an action

(5) Declaration of new words and the symbols associated with them; also, declaration of the associated actions, if any

(6) Delay of actions from the time at which their words appear in the input stream until some later time

(7) Association of an internal symbol to an external word

(8) Variation of the symbols and actions associated with a word as a function of context.†

---

*I am indebted to Allen Newell and Peter Freeman for introducing and familiarizing me with BIP. Also, I wish to thank Robert Yates for assisting me with the LISP implementation.

†This list of capabilities is taken from a working paper entitled "BIP: Basic Interface Package" by Allen Newell and Peter Freeman.

BIP was designed to be a skeleton which can be fleshed out in whatever way is useful for the user. The skeleton itself is completely accessible and is meant to be changed to meet the needs of the individual user.

OVERVIEW

In normal usage the BIP translator will remain in top-level control of the user's system throughout a run. The translator uses an EPAM-type discrimination tree to associate actions and internal symbols with strings of characters from the input stream. These associations are made relative to a syntactic and semantic context. The use of contexts provides an extra dimension of flexibility since the user can easily create new contexts, and change contexts during input to allow the interpretation of any given character string to vary depending upon the environment in which it occurs.

The following definitions will help establish a terminology for our further descriptions:

Character--any character which can be input from a teletype.

Word--a string of characters.

Symbol--the internal data structure associated with a particular word. In the SRI BIP the translator calls the function BIP:CRSYM to create a symbol for a new word. At the time of the call, CHARSK is a list (in reverse order) of the characters which make up the word. The symbol created by the BIP:CRSYM function provided with the package is the atom whose name is the same as REVERSE of CHARSK.

Context--a data structure consisting of any or all of the following: a recognition tree, context mark, action list, and boundary character list. In the SRI BIP a context is a list whose first element is the identifier CONTEXT; the recognition tree is an element of the list whose CAR is the identifier TREE; the context mark is the CDR of an element whose CAR is the identifier CM; the action list is an element whose CAR is the identifier ACTIONS; and the boundary character list is an element whose CAR is the identifier BC.

Boundary character--any character used by BIP in determining the boundaries of a word.

Boundary list--part of a BIP context; it is a list of all boundary characters for a particular context.

Action--a BIP data structure which is associated with a symbol and consists of a priority number, an immediate action, and a delayed action. In the SRI BIP an action is a list whose first element is an integer (i.e. the priority number), optional second element is the immediate action, and optional third element is the delayed action. The immediate and delayed actions may be arbitrary evaluable LISP s-expressions.

Action list--part of a BIP context; it is a set of property-value pairs in which the properties are symbols and the values are the actions associated with them. In the SRI BIP an action list is a list whose first element is the identifier ACTIONS and each succeeding element is a list whose CAR is the symbol and whose CDR is the action associated with the symbol.

Context mark--part of a BIP context; it is used to link BIP symbols with nodes of the recognition tree.

Recognition tree--part of a BIP context; it is a discrimination tree used by the translator for the storage of symbol-definition information. In the SRI BIP each node of a recognition tree is a list whose first element is a one-character identifier (except for the top node which has the identifier TREE as its first element) and whose succeeding elements include the nodes which branch from the node and elements whose CAR is the context mark of some context and whose CDR is a BIP symbol.

Data stack--a push-down stack on which a symbol without an action is pushed after its associated word is recognized in the input stream by the translator. In the SRI BIP the data stack is the list DATASK; CAR of DATASK is considered the top element in the stack, CADR of DATASK is the second element, etc.

Operator stack--a push-down stack on which actions containing delayed actions are pushed to await execution of the delayed actions. In the SRI BIP the operator stack is the list OPERSK; CAR of OPERSK is considered the top element in the stack, CADR of OPERSK is the second element, etc.

Context stack--a stack containing pointers to contexts whose top element is the current context. When the translator enters a context it does so by pushing the context being entered onto the context stack. When the translator returns to a

previous context it does so by popping the context stack until the desired context is the top element. In the SRI BIP the context stack is the list CONTEXTSK; CAR of CONTEXTSK is considered the top element in the stack, CADR of CONTEXTSK is the second element, etc.

THE TRANSLATOR

The translator's flow of control is shown in Figure 1. The input to BIP is a string of characters from some source such as a teletype, external file, or an internal generator. The translator always calls BIP:GETCHAR to get the next character so that it is independent of the source of these characters and the source can be simply switched. The translator leaves to the user the responsibility of selecting the input source.

The recognition philosophy of BIP is to always recognize the longest possible word. Thus, starting at the top of the tree just after a word has been recognized, BIP will work its way down the branches of the tree as long as possible without checking if the character it has just received is a boundary character or not. When it falls out of the tree, that is, cannot find a branch from the current node labeled with the character that it has just received, it checks to see if the current node contains a context mark identical to that of the current context. If not, or if the current character is not a boundary character, it assumes a new word is being defined and proceeds to extend the tree so that it can now recognize it. If there is a context mark and the current character is a boundary character or the previous character was

one, then it knows it has recognized a word and obtains its symbol. If the symbol has an associated syntax action in the current context, it is performed as described below. If it does not have an action, the symbol is pushed onto the data stack. In either case, BIP then begins trying to recognize another word at the point at which the previous word terminates.

In extending the tree to recognize a new word, BIP simply continues to accept new characters until it receives a boundary character. For each new character it adds a new node as a branch from the previous node. When a boundary character is reached, a new data structure (the symbol) is created to associate with the word and a pointer to this structure is stored at the terminal node along with the current context mark; the pointer is stacked on the data stack; and BIP begins trying to recognize another word starting with the boundary character that terminated the new word.

Note that because of the recognition philosophy of BIP it is necessary to have a "quotes context" available to permit the definition of symbols that contain substrings that are symbols and that include a boundary character (once defined, the recognition philosophy permits them to be recognized without any special considerations). For example, we may wish to define the symbols * and *A where * is a boundary symbol. Such a context is supplied as part of the SRI BIP; it has only one boundary character, namely ", and only one syntax action (which is associated with the quote symbol and returns BIP to the previous context). In the above example, suppose we have previously defined * and A as symbols so that they are also boundary characters, and that the action for " in the current context causes the quotes context to be entered.

Then we would write "*A" to define the new word; thereafter, *, A, and *A would be recognized as distinct words.

The recognition and definition of words are lexical actions that are performed by BIP. A user may specify that within any particular context every time a designated word has been recognized a certain syntax action should be taken. This syntax action can be evaluation of an arbitrary function that has been supplied by the user and defined as an action associated with the symbol in the current context. The execution of the action is based on a priority scheme as shown in the flow chart and consists of the execution of an immediate action and possibly an arbitrary number of delayed actions from the operator stack or from the current action (in the order indicated in the flow chart). Since any action (immediate or delayed) is a program, it may do any amount of processing desired; it may work on any of its own data structures or any of BIP's structures (thus effecting BIP's operation) and call any routines whatsoever as subroutines, including the BIP translator itself. In particular, an action may access and alter the data stack (i.e. DATASK) so that the translator acts like a one-stack precedence parser. When the action program is finished, it returns control to BIP which then continues recognizing words in the input stream.

The SRI BIP translator can operate in or out of definition mode. When definition mode is on, all new words are entered into the recognition tree. When it is off, new words are not entered into the recognition tree. A typical use of the mode switch

would be to have it on when actions are being defined for key words (e.g. begin, end, if, then) and then turn the switch off when the only new words being encountered are identifiers and numbers. Since the standard BIP:CRSYM will always return the same symbol name for a given word (i.e. the atom whose name is the same as the word), then it is unnecessary and wasteful to have these words in the recognition tree. Definition mode is defined by the value of identifier DEFSWITCH; T denotes definition mode on, NIL denotes off. The translator initializes DEFSWITCH to T.

Note that any one character word which is entered into the tree is also added to the boundary-character list. This is the only built-in mechanism for defining new boundary characters.

If evaluation of an immediate or delayed action causes the value of BIP:RETURN to be set to T, then the translator will return to LISP with a value of NIL immediately following evaluation of the action. This is the only exit mechanism provided in BIP.

INITIAL CONTEXTS

A base context is provided in SRI BIP which includes the necessary facilities for the user to define the language he wishes BIP to read. When the translator is called, this base context (called BIP:BASECON) is made the current context. In normal BIP usage new contexts are created as copies of existing contexts and then built up incrementally; hence all of a user's contexts can have the facilities included in the base context. BIP:BASECON is defined as follows:

Context mark: MARK

Boundary characters: ⟨blank⟩⟨carriage return⟩⟨line feed⟩
" ' ; . ↑

Actions:

⟨blank⟩ ⟨carriage return⟩ ⟨line feed⟩	Read to the next character which is not ⟨blank⟩, ⟨carriage return⟩, or ⟨line feed⟩.
;	Read to the character following the next line feed. Note, this allows comments to be placed on an input line following a semicolon.
"	Enter the quotes context. The quotes context allows the definition of words containing boundary characters (see the discussion above in the section describing the translator and the description below of the quotes context).
'	Use the READ function to read a LISP s-expression and push a pointer to the expression onto the data stack.
. .	The LISP s-expression named in the top of the data stack is popped off the stack and then evaluated using EVAL.
↑	Exit from BIP with the value NIL.

The quote context (named BIP:QUOCON) referred to above in the description of the translator and in the base context's action for double quote is defined as follows:

Boundary characters: "

Actions: " - return to the previous context.

AUXILIARY FUNCTIONS

The following functions are currently defined in SRI BIP:

BIP:ENCON--a MACRO which takes a context pointer as an argument. The context is pushed onto the translator's context stack and made the current context.

BIP:DEFACT--an EXPR taking no arguments which defines the second element in the data stack as the action for the symbol which is pointed to by the top element in the data stack and does two pop operations on the data stack. The definition is made for the current context. For example, the action for the character ↑ in the context BIP:BASECON could be defined as follows:

'(100 (SETQ BIP:RETURN T)) ↑ '(BIP:DEFACT)..

BIP:CRECON--an EXPR taking no arguments whose value is a newly created context which has the same recognition tree and context mark as the current context and a boundary-character list and actions list which are copies of those of the current context. The user may wish to write other context-creating functions which give the new context a different context mark, a copy of the current context's recognition tree, etc. BIP:CRECON is the only context-creating function provided in SRI BIP.

BIP:DEFCURCON--an EXPR taking no arguments which makes the top context in the context stack the current context.

BIP:RETCON--an EXPR which takes either a positive integer or a context name as an argument. If the argument is an integer

k, then the context stack is popped k times; if the argument is a context name, then the context stack is popped until the named context becomes the top element of the stack. After the popping operations are completed, the top element in the context stack is made the current context.

BIP:SKTOP--a MACRO taking the name of a stack as its argument and returning as its value the top element in that stack.

BIP:SKPOP--an FEXPR taking the name of a stack as its argument which pops the stack and returns as its value the element which was popped off the stack.

BIP:SKPUSH--a MACRO which takes a pointer and a stack name as arguments and adds the pointer to the top of the stack. The value of BIP:SKPUSH is a pointer to the resulting stack.

EXAMPLE

To illustrate the use of BIP we present a set of action definitions which will transform algebraic infix expressions into equivalent LISP s-expressions; e.g. A + B will be transformed into (*PLUS A B). The following are examples from the class of expressions to be translated:

```
A+B+C
(A+B)*C
A+B/-C   .
```

Assuming that LISP has been entered and that the BIP functions have been loaded, the following input sequence will make the desired definitions in a newly created context named INFIX.

```
*(DF DEFBINEXP (L) (BIP:SKPUSH (CONS (CAR L) (REVERSE (LIST (BIP:SKPOP
* DATASK) (BIP:SKPOP DATASK)))) DATASK))

 (DEFBINEXP)

*(BIP)
*'(BIP:ENCON (SETQ INFIX (BIP:CRECON)))..; DEFINE AND ENTER CONTEXT INFIX
*'(6 NIL (DEFBINEXP *PLUS)) + '(BIP:DEFACT)..; DEFINE THE ACTION FOR +
*'(4 NIL (DEFBINEXP *TIMES)) * '(BIP:DEFACT)..; DEFINE THE ACTION FOR *
*'(4 NIL (DEFBINEXP *QUO)) / '(BIP:DEFACT)..; DEFINE THE ACTION FOR /
*'(2 NIL (BIP:SKPUSH (LIST @MINUS (BIP:SKPOP DATASK)) DATASK)) -
*'(BIP:DEFACT)..; DEFINE THE ACTION FOR -
*'(0 (BIP:SKPUSH @(8) OPERSK)) ( '(BIP:DEFACT)..; DEFINE THE ACTION FOR (
*'(8) ) '(BIP:DEFACT)..; DEFINE THE ACTION FOR )
*'(SETQ DEFSWITCH NIL)..; TURN OFF DEFINITION MODE
```

The function DEFBINEXP creates an s-expression to represent a binary algebraic expression. The argument to DEFBINEXP specifies the first element of the created s-expression (the operator), and the top two elements on the data stack specify the second and third elements of the s-expression (the operands). The resulting s-expression is pushed onto the data stack.

The priorities associated with each action provide the desired operator hierarchy. The immediate action for '(' pushes onto the operator stack an action with a lower priority than for any of the operators; the action for ')' is NIL, but its low priority will cause the execution of all delayed actions up to and including the one put into the operator stack by the most recent '('.

Any problems or questions should be directed to Richard Fikes, Room K2090, Extension 4620.

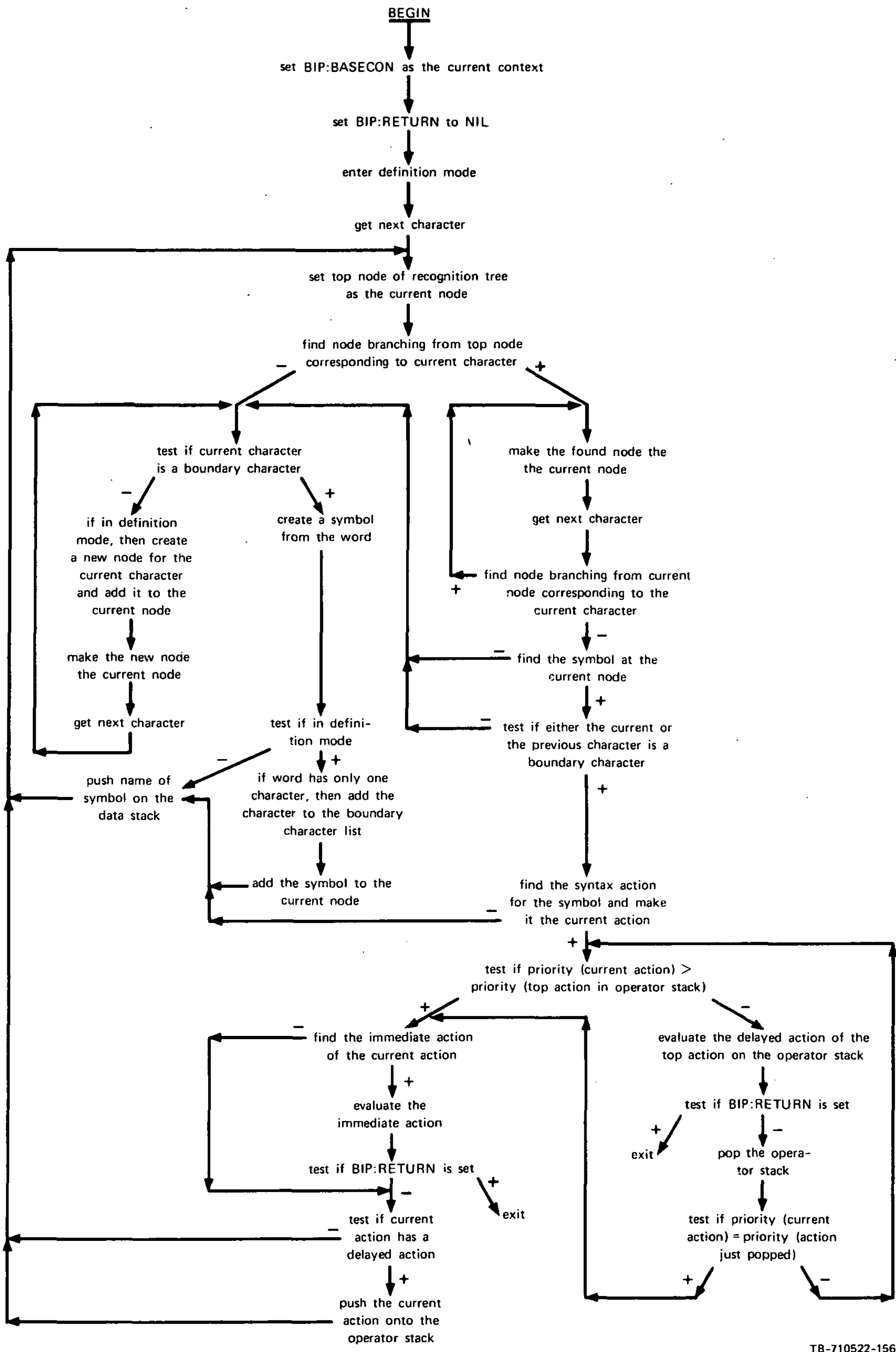


Figure 1 The BIP Translator

Page Intentionally Left Blank

Appendix F

A COST-EFFECTIVENESS BASIS FOR ROBOT PROBLEM-SOLVING AND EXECUTION

Page Intentionally Left Blank

January 1970

A COST-EFFECTIVENESS BASIS FOR ROBOT PROBLEM SOLVING AND EXECUTION

by

John H. Munson

Artificial Intelligence Group

Technical Note 29

SRI Project 8259

The research reported here was supported by the Advanced Research Projects Agency and the National Aeronautics and Space Administration under Contract NAS12-2221.

Page Intentionally Left Blank

A COST-EFFECTIVENESS BASIS FOR ROBOT PROBLEM-SOLVING AND EXECUTION

Introduction

Most, if not all, of the formalized approaches to problem-solving in Artificial Intelligence and robotry to date have been planners exclusively. That is, they deal with a domain represented by an internal computer model, and they plan -- using various methods -- a strategy of actions that is supposed to achieve a desired goal. These approaches all have the inherent property that the plan that solves the problem by the criteria of the planning system has also solved the problem from the experimenter's viewpoint. In other words, the problem domain in the experimenter's mind is the same as that in the system's internal model. When the system reports a solution to a problem, the experimenter can and does check the solution by reviewing it step by step to see if it matches "sound" reasoning done in his own mind. In most or all such systems, the effects of the operators or procedures that may be used to form a solution are entirely known. (Otherwise, "sound reasoning" becomes difficult or impossible.)

Hence, the experimenter who deals with such a system takes a problem known to him, commonly a puzzle or board game, and codifies it in a computer model that is isomorphic to the original. When the system reports its solution to the modeled program, he is happy that it has solved "his problem."

The existence of this isomorphism, however, means that one can only give the system problems that are essentially in the domain of mathematics. They are crisp, exact -- uncertain only if they impinge on Godel's incompleteness theorem. If they purport to reflect real-world, physical problems, the models to date do so only in the most trivial, idealized fashion. If an operator is intended to do something, it will get done.

The complexity and uncertainty inherent in real-world situations and actions are simply not present. This has been the case so far with all the problem formulations given to the problem solver in my group (the QA3 theorem-prover). Of the eleven problems given to the General Problem Solver (GPS) in Ernst and Newell's recent book, only one (the monkey problem) modeled a physical problem, and that in the most idealized terms.

This aspect of the current approaches may be criticized, in my view, as a serious limitation -- in fact, an overriding one -- when the application of problem-solving to robotry is considered. It is fundamental that a robot moving in physical space will be subjected to inaccuracies and uncertainties that are beyond the representational capability of the internal model. Dr. Bertram Raphael put it nicely thus: "the ultimate data base for a robot resides not in the computer but in the actual room around the robot." (The model will, in general, be inadequate in other non-physical respects as well, but this just adds weight to the argument.) Thus, the <u>isomorphism is necessarily broken</u>: <u>an internal problem solution can never be guaranteed to be an external one</u>. Instead, the proof will be in the pudding, and we demand for the solution of our external problem that the robot <u>execute</u> as well as <u>plan</u>, that it act on elements of the plan in addition to thinking them up.

Furthermore, it is not sufficient for the system to think up a plan and then simply turn the physical robot loose on it. Because the outcomes of actions cannot be known for sure, any decent system should monitor the execution of the plan, ready to interrupt if the actual sequence of events diverges from the plan and the attainment of the goal seems unlikely.

There is another requirement. As a step toward reflecting the uncertainty of the physical situation, our desired model will come to include estimates of uncertainty and probability. When this happens, the system will no longer be able to produce a proof that a given sequence of actions will solve a problem; it can only demonstrate a probable outcome. Furthermore, in a long sequence of actions, the probability of following any particular path may become low enough that further contingency planning is not worth the effort. It is more valuable to proceed along the existing portion of the plan and find out what happens before planning further. Thus, the robot must acquire the capability to act before it has completed a plan.

In summary, I have argued that any physical robot is beset by uncertainties surpassing its model, that to experiment with the behavior of such a robot we must deal with execution as well as planning, and that the system will have to decide at times to stop planning and act and at other times to stop acting and plan. To my (admittedly incomplete) knowledge of the AI literature, this topic has not yet been touched -- beyond, perhaps, being given lip service.

A new basis is needed that allows planning and execution to be put on the same footing and related within a decision-making structure. For this basis, I have adopted a broad framework: that of cost-effectiveness, or utility theory. By representing both planning actions and execution actions as elements of strategy possessing costs and effectiveness, we achieve a conceptual framework adequate for the needs noted above. We acquire harmony with the ideas of uncertainty and probability and randomness, and with the concept of progress in an incomplete proof or execution and how to deal with it. We are able to treat sensibly the problems of multiple goals and time-varying goals, and the question of when to quit trying to solve a problem.

In short, I would say that the new framework breaks out of confines imposed by purely deductive processing. GPS took a step in this direction, with its means-end analysis and non-binary difference measures. The path ahead is being explored by probability theorists, proponents of modal logic, and students of the "fuzzy set" concept of Lotfi Zadeh. Arduous though it will be, I feel this is a path we must follow in the development of Artificial Intelligence for use in the real world.

Development of the Framework

I shall now attempt to motivate and develop a cost-effectiveness framework in which to study, describe, and hopefully even implement a robot executive. This framework begins with the notions of states, operators, and transitions in the different worlds (or spaces) viewed by the experimenter and the robot system. It then introduces the idea of effectiveness (positive value or utility, which ultimately derives from the attainment of goals) and the idea of cost (negative value or utility, which has as one of its most important sources the very passage of time), and shows how effectiveness and cost propagate through the state spaces.

(A caution and plea to the reader: I am going to be putting down a fair number of symbols and expressions, most of which won't get wrapped up into tidy equations. These are meant to serve more as shorthand and memory aids than as parts of a "mathematical" treatment. One of the great advantages of the cost-effectiveness viewpoint and the idea of probable outcomes, at least for me, is the feel of what is going on. As I try to work through examples and developments, I can almost see some sort of a mind's-eye robot taking

actions and maybe ending up one way, maybe another. And it seems to help to envision being handed a certain amount of money if a goal is achieved, losing another amount if a passage of time occurs, etc. In other words, intuition and "gedanken-ing" have been my main tools in this development. I shall try to assist your sharing my intuition, through the text. If you can achieve such intuition, and "feel" what is being described, you will understand this framework and, I hope, will believe in it. Long proofs and tedious defenses won't be necessary. If, on the other hand, you don't make your own personal association of meanings with the symbols and expressions that appear, the whole thing will probably look like an exercise in symbol-pushing and you will quit in bafflement and annoyance. The ideas are intuitive; I am trying for persuasion, rather than proof; please try to *feel* the development.)

Robot World-States and Model-States

Our fundamental postulate is that the robot and its physical surroundings are not isomorphically modeled inside the robot's computer. Accordingly, we need to distinguish between W, the external world or environment of the robot, and M, the robot's internal model of the world. At a given point in time, W is in some state W_i and M is in some state M_i.

We can associate W with the experimenter's (presumably omniscient) view of the robot and its real surroundings. For example, the SRI robot currently operates in an environment consisting of a collection of office-type rooms and corridors, largely empty except for doorways, baseboard moldings, an assortment of large, movable wooden boxes, and perhaps some office furniture.

(Incidentally, this type of environment, and tasks such as exploring it, going to particular places, and pushing the movable boxes, will provide the descriptive examples and terminology throughout this paper.) W_i for this robot would consist of knowledge of the room layout, plus specification of the identities, x-y positions and angular orientations of the various objects including the robot. If doors were involved, their state of openness would be included, and so on.

We use M to denote the robot's model, a certain defined body of information inside the robot's computer that represents the robot program's knowledge of its situation. Given the present state of the art, M will tend to present a very simplified and stylized reflection of W. (The very reason, of course, why we and other researchers set up such clean environments for our robots is an attempt to create worlds so simplified that our models can even begin to represent them.)

We will take the view that the only information about the robot's condition that the robot program can directly access is that in M_i. If the program wishes to learn something from W_i, it must invoke some sensing operator or action operator, which will cause a state transition in M-space and possibly also in W-space. The new information about the robot's environment that is available to the robot program is that which appears in the new model-state M_j.

Thus, the act of perception is represented by an explicit operator, and the vagaries of perception can be treated by the probabilistic transformation structure that we shall develop below. Handling perception thus is part and parcel of our recognition that the world and the robot's model of it are two different things.

In our present plans for the SRI robot, the model M will consist of ordered n-tuples of information. The following unofficial but illustrative sample should be largely self-explanatory.

(X	ROBOT	37.6)		(X	OBJ_1	50.0)
(Y	ROBOT	- 5.0)		(Y	OBJ_1	20.0)
(θ	ROBOT	47.0)		(OBTYP	OBJ_1	BOX)
(IN	ROBOT	OBJ_2)		(COLOR	OBJ_1	RED)
(OBTYP	OBJ_2	ROOM)			. . .	
(NAME	OBJ_2	JOHN'S)		(DOOROF	OBJ_2	OBJ_3)
(NAME	OBJ_2	K2060)		(STATUS	OBJ_3	OPEN)

(NORTHWALL OBJ_2 43.3), etc.

(The reader should not be dismayed if this sample seems to raise many questions of representation. The problem of representing real situations is an extremely difficult one, which can be expected to occupy AI researchers for decades. In fact, I consider this problem -- which can also be stated as that of developing a machine epistemology for AI -- to be the central and ultimate challenge of AI research. The sample shown above is presented only for the purpose of establishing some intuitive material for future examples and discussions.)

Operators, Probable Outcomes, and Estimates

The robot has available to it a certain repertoire of operators: for example, turn, move forward, and many more. An operator, depending on its type, may or may not cause a change in W (in other words, a transition from some W_i to some W_j). Similarly, an operator may or may not cause a change in M.

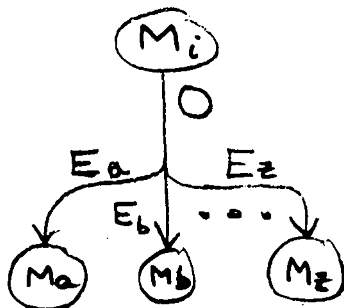
In a purely non-physical computer program, and also in human thought, the distinction between the operator and the result it achieves tends to be blurred. On a chessboard, for example, "pawn-to-King-four" names the action and the result simultaneously.

When dealing with a robot, by contrast, we must differentiate between the operator, the change it produces in W, and the change it produces in M. When we speak of an operator as "move ahead four feet" or "go to x = 20.6, y = 6.7," we are actually naming the operator according to its nominal, or desired, result. The real robot will most certainly not move ahead four feet exactly in W. Given a clear path, it may move ahead a random distance described by a Gaussian distribution with a mean of 3.92 feet and a standard deviation of 0.2 feet (and turn and drift sideways randomly as well). Given an obstacle in the path, the robot may stop at any point. What happens to M, moreover, depends not only on what happens to W but on the system that feeds information from W back to M.

In some respects (such as whether there is an obstacle in the path) we may consider that the experimenter knows exactly what will happen. In other respects (such as the random stopping distance described above) the experimenter does not know what will happen, and we will conceptually describe his (or our) state of partial ignorance with a probability density function.

W_j or M_a may be taken to represent all outcomes of a move-four-feet operation when no obstacle is encountered, and so on. Of course, these groupings are approximations, and how to handle the compounding of such approximations is an unsolved problem. Finally, it is possible to view the probabilistic branching as "playing a game with nature." One chooses an operator O, and nature responds with a resulting state. Some aspects of AI research in game-playing may be applicable. However, nature here plays probabilistically, not to maximize value, as is assumed in classic game theory and in most research.

Now the robot's executive program, much more than the experimenter, will be burdened with ignorance about the outcomes of operations. Thus, the program needs to estimate the probable outcomes,and its estimates can be represented by a similar diagram:

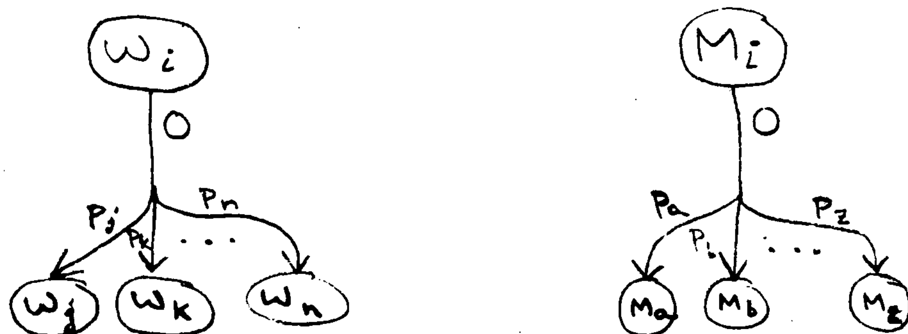


This diagram is drawn only in M-space because the program has access only to M-space; it never "sees" W-space directly.

The various estimates made by various robot programs may range all the way from simple-minded assumptions that the desired result will always occur to highly sophisticated calculations involving information from the model, learning from past experiences, and so on. The estimates may be quite accurate or totally fallacious in any given situation. They may appear as probability calculations, or in some other guise. In any case, we <u>conceptually</u> view any assumption made by the program about the outcome of an action as a probability estimate of this form.

(Of course, these probability functions will often be extremely complex and beyond calculation. I take the view, however, that the probability concept is both a fruitful and a philosophically valid one (two different things) for representing partial knowledge in a decision-making situation. Throughout this development there will be many such functions named and left unexamined. Finding workable approximations or equivalent methods is the task of research. This paper aims to create a framework, not fill in all the blanks.)

We can represent diagrammatically the idea that an operator, applied to a state of either W or M, will give rise to different results according to some probability:



Several points may be noted. First, we have emphasized the separateness of W-space and M-space. Second, although not shown, O may represent an instance of an operator, selected by parameters (such as "move <u>four feet</u>"). Third, the outcomes and their probabilities will generally depend on both O and the initial state, and perhaps on other variables in the robot system through their implicit relationship with O. Fourth, one will often in practice use a grouping of final states: for example,

Goals, Payoffs, and Time

A goal for the robot (synonymous with a problem to be solved) is represented as a state, or set of states, for the robot to achieve. Often a partial state description is given, such as telling the robot to go to a certain place, with the understanding that any state satisfying the stipulation achieves the goal. For visibility, we shall often show a goal state in W-space or M-space as Ⓖ .

It should be noted that a goal can be specified to the robot system only in M-space, since the system is not directly cognizant of W-space. Overlooking this fact (tantamount to re-establishing the isomorphism between W and M) has unfortunately helped lead some to talk of M-space specifications, for example "go to Room K2060," as if they were unique problems in W. In fact, there are as many such problems as there are robots, worlds, and starting states -- in other words, contexts or "frames" for the goal specification.

Associated with each goal -- and we shall be quite happy to accommodate multiple goals -- is a payoff U_G measured in units of utility. U_G represents the value to be realized by the achievement of the goal.

In the simple case, U_G is merely a constant. However, one could envision more complex goal specifications, containing subclasses with differing U_G's depending on the route taken to the goal, resources used, etc. (We shall see that such factors are often better expressed as costs on the way to the goal.)

Most importantly, and requiring some discussion, utility is related to time. A quick solution to a problem is considered better than a slow one, and must be made to appear so to the robot system. We all know that the familiar "exhaustive solutions" that take longer than the age of the universe are not solutions at all for our purposes.

In some instances the goal might have an explicit time constraint, such as "find a red cube within five minutes," and then the payoff would be explicitly time-varying. Usually, however, it will be natural and effective to let the payoff of the goal(s) be fixed and associate a negative utility, or cost, with the passage of time.

This cost of time is not intended to reflect the expenditure of power or other resources by the robot system; these can appear later as explicit costs in the formulation. The cost is intended to reflect the basic fact that the employment of any person or machine to perform a function generally has a cost per unit time; hence, a faster system is a better system. Experimentation with a robot system that has any capacity to schedule its own behavior should reflect this fact. Even if the model of a useful robot were discarded as a reason, the value of the experimenter's own time would lead to the establishment of such a cost.

Now it is true that most existing problem-solvers do not associate any cost with time. They work on a single problem; the problem at hand is the entire world to them; they pursue it until they succeed or demonstrably fail or the experimenter cuts off the run. But how can such a system arrange intelligently to handle multiple, coexisting goals with different priorities?

Only by being able to schedule itself can such a system perform, and this requires estimating the cost, in time, of its actions and relating the cost to the utility of its goals. Our framework will provide a system that can drop one goal, or line of action, if its prospects become bleak or another more promising one is injected. Furthermore, the system can terminate its activity by deciding that a goal is no longer worth working on. (The reader may suddenly picture himself confronted with a stubborn robot that refuses to

work on a perfectly good problem that he wants worked on <u>in any case</u>; if this happens, either the robot's estimates or the assigned ratios between goal payoff and time cost are wrong. The dubious reader is invited to ponder this for himself.)

In a later section we will continue the discussion of utility, showing how it can be "backed up" from state to state, using the costs and probabilities associated with operators. But first we must examine the role of planning in the robot system and determine the space (neither W nor M) in which the system will be considered to operate.

<u>Planning and the Knowledge-Space S</u>

In the discussions above, we have provided settings for the robot's active and perceptual operations. Actions are operators that change the state of the world W, and very likely of the world-model M; perceptual operators are certain ones involving the physical robot but devoted primarily to updating information in M. We have described how the non-trivial relation between an action and its outcomes is encompassed by describing the action with probabilistic state transitions, and the non-trivial nature of perception is handled by describing their effects on M the same way.

It remains to provide a setting for the planning (or "thinking," or cognitive) operations of the robot program. In doing so, we propose to limit sharply the scope of the model M, to <u>that information which directly represents a model of the world-state at a given instant in time</u>. Information generated or obtained by the robot program above and beyond what is in M will be represented in a new space, which we shall denote as S. To illustrate, the

knowledge that the robot is in $Room_1$ is an element of M, but the knowledge or deduction that, if the robot invokes operator O it may then be in $Room_2$, is outside of M and is an element of S.*

In fact, the intuitive definition of S is that it is the space of states of knowledge of the robot program. Thinking or planning activities of the robot will generally cause a change of state in S, by adding knowledge to the system. Execution actions of the type discussed previously will in general advance M in time, thus rendering some knowledge in S obsolete and pruning the knowledge tree. Planning and execution then become related as alternative operators that can cause transitions in the new space S. We will in turn be able to discuss utilities and probable outcomes in S, thereby arriving at a rational, cost-effectiveness based framework that includes and relates robot planning and execution.

An Example of a Knowledge-Space

We will illustrate the structure and the use of the knowledge-space S by means of an example drawn at the simplest possible level. Although I believe this example is authentic in spirit, I do not claim that it is a finished product nor that it truly represents any realistic robot system. It is stripped down to the bare bones, and its purpose is to illustrate a space S as plainly as possible.

Consider a robot that is in a world-state W_o and model-state M_o, and is to achieve a goal G. (G is a state specification in M.) For our example, we will assume that the robot is at some point within a single closed rectangular room, that the room contains some boxes, and that the goal is

*Note added in proof: Thus, goals such as "explore" and "visit all rooms" are inherently outside of M. We should take the viewpoint (which the paper currently does not) that goals are state specifications in S, only some of which happen to correspond directly to states in M.

to have the robot at some other specified point in the room. The reader can visualize the world-state W_o for himself, and he can take the sample model given in an earlier section, suitably completed, as representing M_o. The goal specification is

(X ROBOT X_G)

(Y ROBOT Y_G)

where X_G and Y_G are the co-ordinates of the goal point.

We explicitly separate planning and execution. We assume that the robot program has available to it two planning operators, A and B. Planner A, if invoked while the model is a state M_o, <u>may or may not succeed</u> in producing a plan (denoted AA) for achieving the goal. If a plan AA is produced, and if it is executed while the model is in state M_o and the world is in state W_o, the plan in turn <u>may or may not</u> achieve the goal G. Similarly, planning operator B under the same conditions may or may not produce a plan BB, which in turn may or may not achieve G.

We make two simplifying assumptions. First, we assume that the execution of a plan proceeds as an unbroken unit and hence may be considered as a single operator for our purposes. Second, we assume (somewhat unrealistically) that if an execution operator fails to achieve the goal, it leaves the world in state W_o and the model in state M_o. Thus, M_o and G are the only model-states involved in our example.

(Although it is not strictly necessary for the development, the reader may find it helpful to carry a mental picture such as the following. Planner A checks whether the straight-line path from the robot's position to the goal is clear in the model. If so, A generates a plan AA which consists of a

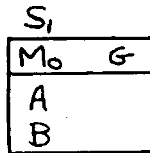
simple turn and move forward to the goal. Planner B is a more complex algorithm for route finding among obstacles. (Various algorithms, such as Moore's method and the tangent-point graph procedure, have been investigated in our group.) B will not produce a plan BB if it thinks that no through path exists. B is more "sophisticated" than A. B will generally find a plan whenever A does, but that might not be the case if B demands a greater tolerance for skirting obstacles. Given a model that accurately reflects the world, both planners will produce only successful plans; given an inaccurate model, either one might produce the higher percentage of unsuccessful plans. We simplify further by identifying modeled success with external success: if the planned moves go to completion without an unexpected bump, we assume that the goal conditions are achieved in the model and that the robot moves close enough to the physical goal in W to satisfy the experimenter. If a bump occurs, the robot retraces its path and leaves the world in state W_o and the model in state M_o.)

Considering now the beginning of our example experiment, we observe first that the execution operators potentially specified by plans AA and BB cannot be chosen by the system because it has not thought of them yet. (If this seems somewhat foreign, it is because we are conditioned to the type of system described in the introductory section, in which successful planning implies and even constitutes successful execution.) The only operators potentially capable of changing the state of the system in S-space, that are available at the outset, are A and B.

We take the view that the system always knows, at a primitive level, which planning operators are potentially applicable and which have already been tried, in any given state S_i in knowledge space. That is, we assume that those calculations are built into the system and done without cost

whenever needed, rather than themselves being subject to the cost-effectiveness mechanism. This point will be discussed fully later.

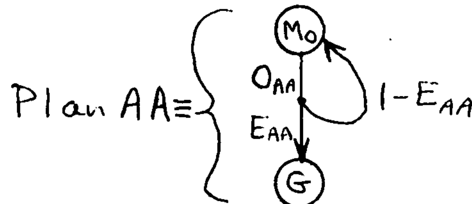
In our example, we may represent the starting state of knowledge S_1 thus:



This informal diagram means that, in the state S_1, the current state of model-space is M_o and the goal is G. The system currently has available to it planning operators A and B, and no execution operators.

Suppose now that the system chooses to invoke planner A. Invoking A will cause a probabilisitic state transition in the knowledge space S, with two possible outcomes, according to whether or not A produces a plan.

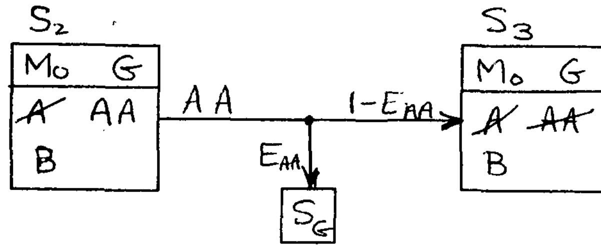
Let us examine the plan that A might produce. Viewed in M-space, the plan has the form shown in an earlier section:



Put in English, the plan is something like this: "While in state M_o, invoke operator O_{AA}. With estimated probability E_{AA}, the goal G will be achieved in M-space. Otherwise (in this example) the state of M will be unchanged." (E_{AA} is the program's estimate; it may, of course, not match our own "omniscient" value P_{AA} for the probability of success of the operator. A simple planner may put $E_{AA} = 1$, while we know very well that the planned action will fail sometimes.)

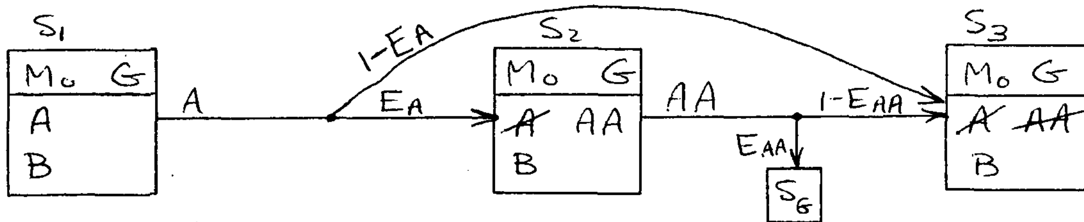
Now let us consider the same plan from the viewpoint of S-space. Here it appears as a new execution operator AA which can be selected by the system. Since there is something new relative to the starting state S, the system

must be in a new state of knowledge S_2. We can draw the S-space view of the plan thus:



The interpretation of this diagram is that the application of AA while in state E_2 is estimated, with probability E_{AA}, to achieve the goal state S_G. (S_G is defined in this simple example as any state of knowledge in which the model achieves state (G). By our previous simplifying assumptions, the problem is then solved, and nothing else matters.) If AA fails to achieve the goal state, the system will then be in a new state of knowledge, S_3, in which AA has been exhausted. The appearance of crossed-out operator symbols is a reminder that they are exhausted relative to the state of knowledge in which they appear.

We may now include the starting state S_1 and the outcomes of the planning operator A in our diagram:



With estimated probability E_A, planner A will produce the state of knowledge S_2 in which the plan AA exists. Otherwise, A is exhausted without producing an AA, yielding a state equivalent to S_3.

By applying the same considerations to planner B and its plan BB, we obtain the complete "three-by-three" S-space transition diagram for our (simple!) example, which is shown in Fig. 1. This diagram shows all the possible states of knowledge, and the applicable operators at each state.

In state S_9, both avenues A-AA and B-BB have failed, and the problem is not solvable by the system.

In this example, the execution operators AA and BB bear a one-to-one relationship to the plans described by transition diagrams in M-space. In a more complicated system, this one-to-one relation might not hold. The essential idea is that an operator in S-space is <u>anything</u> that changes the state of knowledge of the system, whether it modifies the plan structure by "thought" (adding, modifying, re-evaluating, or abandoning plans) or by "execution" (which will in general prune part of the planning structure and will in any case exhaust the execution operator relative to the current state).

The Analysis of Payoff in Knowledge Space

It is not possible to assign absolute utility values to the states of a graph such as that of Fig. 1, because of the existence of closed loops with non-zero cumulative costs around the loops. (Another way of looking at it is that states such as S_3 can be reached at different times by different routes, hence with different time costs.) Instead, we must deal with incremental amounts of utility, namely, payoffs. Payoffs may be established for states in several ways.

First, payoffs may be assigned to terminal states, in which the experiment ends. In our example, we assign payoff U_G to the goal state S_G in S-space, and we assign a payoff of zero to the state S_9, in which the experiment must be terminated without success.

Second, payoffs may be backed up to a state by the use of two rules. Rule 1 states that the expected payoff of applying operator O in state S_i, denoted U_i^O, is the average of the payoffs U_j of the possible outcomes of O, less the costs of the transition, weighted according to estimated probability. Thus,

$$\text{Rule 1:} \quad U_i^O = \sum_j P_j(U_j - C_j),$$

where P_j represents an estimate of the probability of reaching state S_j by invoking operator O in state S_i, and C_j is the cost along that branch. (If the C_j's are all equal, they can be represented by a single C, and the formula becomes $\sum P_j U_j - C$.) Note that if the robot system rather than the experimenter is doing the estimating, its probability estimates E_j are used for P_j.

Rule 2 states that the expected payoff of a state is the maximum, over all operators applicable in that state, of the payoffs for each operator. Thus,

$$\text{Rule 2:} \quad U_i = \max U_i^O, \text{ over all O}$$

Finally, payoffs may be evaluated (either for a state or for the application of an operator to a state) by an evaluation mechanism that uses the data describing the state and/or operator in question.

The methods above are fully analogous to those of game-playing programs and game theory, with the difference (as noted earlier) that the "opponent" behaves probabilistically rather to maximize his own utility. The branch points for probable outcomes of operators are the analogs of the alternate plies in a game tree, at which the opponent moves. In fact, I believe that a variant of game theory that deals with a "probabilistic opponent" has been developed under the name of "expectamaxing," analogus to "minimaxing."

Let us see what would be required to back up payoffs throughout the state space of Figure 1 from the terminal states. Figure 2 shows the space again, with the costs and expected probabilities of outcomes listed for each operator. (We are assuming constant costs.) Using Rules 1 and 2, the various utilities are calculated as follows. (The utilities for S_4, S_7, and S_8 are obtained from those for S_2, S_3, and S_6 by interchanging A's and B's.)

$$U_6 = U_6^{BB} = E_{BB}\, U_G - C_{BB}$$

$$U_3 = U_3^{B} = E_B U_6 - C_B$$

$$U_5^{AA} = E_{AA} U_G + (1-E_{AA})\, U_6 - C_{AA}$$

$$U_5^{BB} + E_{BB} U_G + (1-E_{BB})\, U_8 - C_{BB}$$

$$U_5 = \max(U_5^{AA}, U_5^{BB})$$

$$U_2^{AA} = E_{AA} U_G + (1-E_{AA})\, U_3 - C_{AA}$$

$$U_2^{B} = E_B U_5 + (1-E_B)\, U_8 - C_B$$

$$U_2 = \max(U_2^{AA}, U_2^{B})$$

$$U_1^{A} = E_A U_2 + (1-E_A)\, U_3 - C_A$$

$$U_1^{B} = E_B U_4 + (1-E_B)\, U_7 - C_B$$

$$U_1 = \max(U_1^A, U_1^B)$$

A complete backup of payoff would thus require knowing the costs C and expected probabilities E of every operator. Within the limitations inherent in the use of these quantities, complete look-ahead (as it is called when viewed from the starting state at which a decision must be made) provides an optimum rational basis for decision-making in the face of uncertainty.

In practice, of course, complete look-ahead is generally impossible. In board games, it is often feasible to look ahead exhaustively through a few levels of branching, and to do so with precision because all options of self and opponent are known. At the tips of the look-ahead tree, unless they are terminal, <u>evaluation</u> is employed to establish payoff utilities, which are then backed up.

In the case of the robot state diagram, look-ahead is likely to be abandoned much sooner. The introduction of costs and probabilities, together with the knowledge that we will never in practice determine most of them beyond an educated (or uneducated) guess, will undoubtedly induce us to abandon look-ahead at an early point--even at the starting state! -- and rely on evaluation of the available operators.

Thus, we are led to consider the means by which operators may be evaluated. Most simply, the payoff of an operator may be taken as a constant, or, better, a constant ≤ 1 times the payoff of the goal in question. For the purpose of decision-making without look-ahead, it would suffice even to rank-order the available operators. Any program that has a fixed order of application of its operators is in effect rank-ordering them. A more powerful technique is to evaluate the expected payoff of an operator in the context of the current state. (Deciding whether an operator is

applicable to a given state is an extreme example of this.) In our robot example, the evaluation of the expected payoffs of the planners might reasonably be made to depend on the distance from the robot to the goal, the count of boxes in the room, and so on.

Conceptually, the spectrum of possible evaluators reaches all the way to those that would simulate every action of the operator being evaluated. Such an evaluator would of course be worse than useless, because it would be as complex, bulky, and costly to run as the operator it copies. Generally, the idea is for a simulation to be an inexpensive approximation to the simuland. But what if, as may often be the case, there appears to be no worthy approximation to the operator simpler than the operator itself? I offer, as an interesting topic to explore, that of letting some of the operators in the system act as their own simulations. For example, if the routines that cause motors to turn, etc. on the physical robot were temporarily replaced by dummy simulations, it would be possible actually to call an execution operator in a kind of Gedanken mode, and use the outcome of this Gedanken-experiment to evaluate the operator. The evaluation would automatically occur in the context of the present state of knowledge; the current model M, and so on. Because physical motions of the robot would be avoided, the Gendanken world could run faster than real time and thus meet the necessary requirement that the evaluator be less costly than the operator being evaluated. Furthermore, once the Gedanken world were created, any higher-level operator could be run in Gedanken mode without further ado.

Hierarchical Organizations of Spaces

In our example used in the previous sections, we portrayed a two-level hierarchy of spaces, namely, the model space M and the knowledge space S. We concentrated on S, in an attempt to show how planners, plans, and executors dealing with M could be related in a coherent framework from the viewpoint of S-space. We can picture S-space as a kind of higher-level space, or meta-space, relative to M.

Our postulated monitor, or S-space program, has a cost-effective view of each of the lower-level operators it can invoke, such as a planner. Each planner, in turn, could be viewed as having a cost-effective view of the operators that it can choose in the construction of a plan. Whether or not a given planner is *actually* programmed in this fashion is another matter. I am claiming that the cost-effective framework is, first, a valid and all-encompassing conceptual one for treating any decision-making system, and second, a framework in which planners at any level *could* be coded. I am not claiming that it is desirable or practical to do so. In fact, in view of the rather tedious and abstract nature of state-space expansions, it is probable that lower-level operators will be programmed more in pragmatic and specialized fashion than as explicit cost-effectiveness calculators.

At the higher level, that of the monitor program that deals with S, the chances are better for practical realization of a cost-effective calculator. But it must be borne in mind that the monitor itself is subject to design considerations, that our simple example tended to gloss over.

Even in our example, there was a choice (which we discussed but did not make) of how much look-ahead the monitor should perform. At one extreme, the program could look ahead all the way to the terminal nodes, as we

ourselves did in Fig. 2. At the other extreme, the program could perform no look-ahead. Finding itself at a given state of knowledge, the monitor could simply evaluate all the available S-space operators and choose the one with the highest estimated payoff.

We also assumed, in the example, that the monitor could always enumerate the available operators. This might not be the case in practice, and we might have to describe for the monitor a strategy for choosing which operators to choose for consideration by look-ahead and evaluation.

A further assumption in our example was that the monitor uses the probability estimates E that the planners generate. This assumption is not necessary; the monitor could in fact modify the E's or make its own entirely different estimates. The monitor's behavior could then be analogous to that of a supervisor who didn't take on faith whatever his subordinates told him about the projected success of their plans.

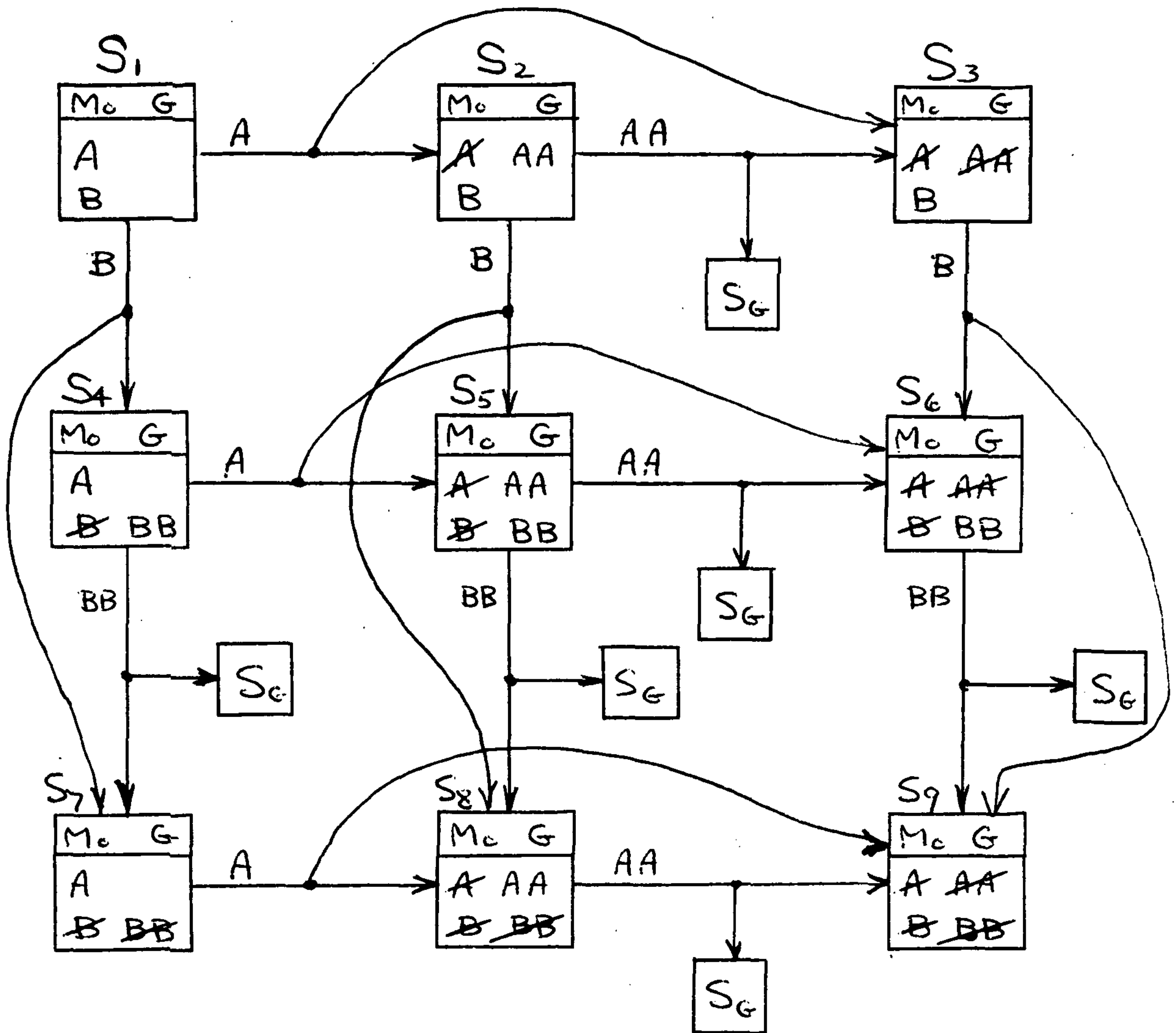
Another possible variation of the monitor is to allow the possibility of quitting at any point. In S-space, this amounts to including, at every state, an available operator that has zero cost and that always leads to a terminal state with zero payoff. As a consequence, the monitor will never proceed past a point at which the other operators all have negative expected payoff. This would seem to be a pretty refinement in an experimental robot.

From the foregoing paragraphs, it should be clear that the design of the S-space monitor is by no means fixed. In fact, there is an infinite family of cost-effectiveness-based monitors, not to mention all the other types of "monitors" that could be used to control the use of the lower-level operators. Then, we can envision a given S-space monitor as being merely a kind of higher-level planner, and we can picture a collection of such monitors as being subject to regulation by a "meta-monitor" operating at a higher level.

The meta-monitor might or might not be expressed in cost-effectiveness terms; if it is, we can describe it as working in a higher meta-space S', for which the available operators are the S-space monitors and, possibly, direct use of lower-level planners and executors. The S-space programs might continue to exist as true monitors (in that they can invoke execution operators), or as Gedanken-monitors, or as high-level planners only, with the decision to invoke executors left to the meta-monitor.

It is thus clear that, as we add additional levels to the control hierarchy, richer and richer structures occur. Furthermore, it should be evident that there is no end to the number of levels that can be added (conceptually at least), and that from structure to structure the question of the roles played by the various levels and operators is finally a matter of choice, terminology, and concept. It is the task of robotry research to develop and experiment with such structures, toward the twin goals of achieving understanding of them and creating useful systems. This paper has offered little specific guidance for this task, but it has established the necessary conceptual framework for robot systems that act as well as plan, and has suggested how operators based on the idea of cost-effectiveness could be used at various levels within the system.

FIGURE 1



S – space transition diagram for the example
(Probability estimates not shown)

FIGURE 2