Final Report

Covering the Period 1 June 1971 to 31 July 1973

RESEARCH IN ADVANCED FORMAL THEOREM-PROVING TECHNIQUES

By: B. RAPHAEL, R. FIKES, and R. WALDINGER

Prepared for:

NATIONAL AERONAUTICS AND SPACE ADMINISTRATION
HEADQUARTERS
600 INDEPENDENCE AVENUE, S.W.
ROOM 607
WASHINGTON, D.C. 20546
Attention: MR. CHARLES PONTIOUS

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

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

Final Report *August 1973*

Covering the Period 1 June 1971 to 31 July 1973

RESEARCH IN ADVANCED FORMAL THEOREM-PROVING TECHNIQUES

By: B. RAPHAEL, R. FIKES, and R. WALDINGER

Prepared for:

NATIONAL AERONAUTICS AND SPACE ADMINISTRATION
HEADQUARTERS
600 INDEPENDENCE AVENUE, S.W.
ROOM 607
WASHINGTON, D.C. 20546
Attention: MR. CHARLES PONTIOUS

CONTRACT NASW-2086

SRI Project 8721

Approved by:

B. RAPHAEL, *Director*
Artificial Intelligence Center

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

Copy No. 46

ABSTRACT

This report summarizes the results of a three-year project aimed at the design and implementation of computer languages to aid in expressing problem solving procedures in several areas of artificial intelligence including automatic programming, theorem proving, and robot planning. The principal results of the project have been the design and implementation of two complete systems, QA4 and QLISP, and their preliminary experimental use. QA4 has been documented in detail in a previous technical report.*[1] This report contains a description of how both QA4 and QLISP have been used; the Preliminary QLISP Manual is attached as an appendix.

*References are listed at the end of this report.

CONTENTS

ILLUSTRATIONS

ACKNOWLEDGMENTS

No useful programming system can be developed without the cooperation of users who are willing to exercise each new experimental release, thereby helping to debug the system and provide valuable criticism to help in designing the next improved version. In addition to the authors of this report and its appendix, the following people helped us by using various undebugged versions of QA4 and QLISP: J. F. Rulifson, N. J. Nilsson, K. Levitt, B. Elspas, C. C. Green, I. Greif, M. Stickel, D. Lenat, D. Shaw, T. Garvey, and A. E. Robinson. The research applications for which QA4 and QLISP have been used are supported at SRI by the following sources: ARPA, under contracts DAHC04-72-C-0008 and DAHC04-72-C-0009; NSF, under grant GJ-egl46; and ONR, under contract N00014-71-C-0294.

I INTRODUCTION

A. Background

During the late 1960s the SRI Artificial Intelligence Center was engaged in a variety of research projects in the areas of automatic theorem proving,[2] question answering,[3] problem solving,[4] and robot systems.[5] One of the themes of this collection of projects was the use of a theorem proving system, QA3,[6] as the single deductive mechanism applicable to a range of other project goals. This approach had obvious appeal: Isolating the deductive component as an identifiable module simplified the conceptual structure of the various systems, and one could hope that improvements to the theorem proving module could cause the performance of all the other systems to improve simultaneously.

In practice, our approach had limited success. We did indeed develop demonstration systems for question answering and robot planning that were at least comparable in ability to any others in existence at the time. However, these systems could not be extended easily to handle larger, more realistic problem domains. A major bottleneck seemed to be the limited expressive power of first-order predicate calculus, the formal language used in our theorem proving system, for encoding the complex knowledge needed by the computer to solve harder problems.

In June, 1970, we began work for NASA on advanced formal theorem proving techniques. Our original plan was to develop a richer logical calculus to substitute for our QA3 system. We quickly discovered, however, that the problem of developing an automatic theorem prover for a system of logic--such as a higher-order predicate calculus that has the expressive power needed for interesting artificial intelligence research--

is itself a difficult problem of artificial intelligence (AI). Moreover, we decided that work on this class of problems was severely handicapped by inadequacies of the then-existing programming systems. Just as list-processing languages freed AI programmers from a mass of bookkeeping details and enabled a spurt of major research results a decade ago, we felt that new languages with a variety of novel built-in features we were just beginning to understand could promote a major step forward in AI progress today. Therefore, the NASA supported effort quickly turned from the study of theorem proving techniques per se to the development of software systems that we felt were prerequisites for major research progress in a broad spectrum of AI activities.

This report describes the results of three years' work on the development and experimental use of new programming languages for AI research.

B. New Programming Languages for AI Research

A new generation of programming languages is now becoming available. These languages, which have many features in common, were developed more or less simultaneously by several AI research laboratories, in response to largely independently-discovered needs. A tutorial review of some of these languages is given in Ref. 7.

The special features common to most of these languages can be described under the following headings.

(1) Data Types and Memory Management. In addition to the now familiar symbolic data types, lists and strings, the new languages allow manipulation of such constructs as sets, trinary associations, and formal theorems. They also usually provide large, permanent data stores, with efficient built-in storage and retrieval procedures.

(2) Control Structures. The major innovations are automatic backtracking, programmable changes of control environment ("contexts"), pseudoparallel processes, and automatic condition monitoring devices ("demons").

(3) Pattern Matching. Complex pattern matching functions can be used for verifying the structure of the data, binding variables to subexpressions of the data, and selecting appropriate programs to execute.

(4) Deductive Mechanisms. Built-in search and default inference procedures are useful features of these systems.

(5) Special Operating Environment. Opportunity for intimate on-line interaction and powerful debugging facilities greatly enhance the effectiveness of a programming system.

The following languages are the major representatives of this new generation.

(1) SAIL,[8] a combination of ALGOL and an associative information retrieval system called LEAP, extended to meet the needs of the Stanford AI Laboratory.

(2) PLANNER,[9] a language developed at MIT in connection with the concept of procedural representation of knowledge.

(3) CONNIVER,[10] a successor of PLANNER that gives the programmer more direct control and responsibility with respect to program direction.

(4) POPLAR,[11] an extension of the POP2 language developed at the University of Edinburgh School of AI.

(5) QA4 and QLISP, the languages developed at SRI under the project reported here.

We believe that QLISP contains most of the desirable features of this new generation of languages in a well designed system framework, and that it has an excellent chance of becoming the most important software tool for the next few years' AI research.

C. Overview of the Project

During the first year of this project, 1970-71, our ideas about needed language features were crystallizing. These ideas were documented in a variety of memos and papers that are summarized in an annual report.[12] The second year, 1971-72, was primarily devoted to the implementation, refinement, and initial use of a major new programming language, QA4. That system, along with its motivation, philosophy, and uses, was documented in a 360-page technical report submitted in November of 1972.[1]

Since late 1972, our primary activity has been the development of a second generation of our new language. The resulting system, called QLISP, will contain all the desirable features of QA4; but, they will be packaged in a much more efficient and more usable framework. We expect QLISP to be the major programming system for AI research at SRI for years to come.

Although QLISP still needs some significant modifications, a preliminary version is now operational. The Preliminary QLISP Manual is attached as an appendix to this report. We plan to make this manual available separately as a Technical Note in order to promote experimentation with the system.

Even though QA4 and QLISP are still incomplete experimental systems, we have already used them effectively for substantive research, partially supported by other projects, in the general areas of robot planning, theorem proving, and automatic programming. The next section of this report describes some of these applications.

II. APPLICATIONS OF QA4 AND QLISP

During the past year QA4 and QLISP have been used as tools in the exploration of design ideas. With the aid of QA4 and QLISP we have found it much simpler to create trial implementations of ideas and to run experiments with those implementations than we have in the past. QA4 and QLISP contain the pattern matching, expression manipulation, modeling, and control that are common to much of our work, and therefore a new program design idea can be quickly implemented without having to expend large amounts of effort rebuilding these basic mechanisms. Some of these implementations and features of QA4 or QLISP that were particularly useful are discussed below. This discussion assumes that the reader is familiar with the basic QA4 notations, such as inverse quote mode, and concepts, such as the use of "demons." These notations and concepts are described extensively in Ref. 1.

A. A Deductive Retrieval System

In our work with robot planning and more recently in our work with a system for providing guidance to a man doing some maintenance task, our programs need to maintain a model of the physical environment in which the robot activity or maintenance operation is taking place. Our planning, monitoring, and advice-giving programs ask questions of this model to determine how they should proceed. Hence, we need a question answering subsystem that can interface with the model. This subsystem should provide a deductive capability for dealing with questions whose answers are not explicitly stored in the model but whose answers can be deduced from information that is in the model. A new design for such a subsystem was tested during the last year with an implementation in QA4.

This design is an augmentation of the QA4 model retrieval statements EXISTS and INSTANCES. These statements retrieve from the model one or all true instances of a given expression containing variables (i.e., a pattern). For example, the statement (EXISTS (ON ←X DESK1)) could retrieve from the model one object that is on DESK1, and the statement INSTANCES (ON ←X DESK1)) could retrieve all objects that are on DESK1. Now, consider as another example the statement (EXISTS (TOP:CLEAR DESK1)). This statement will look in the model for the expression (TOP:CLEAR DESK1) to indicate that no objects are on DESK1. If that expression does not have value TRUE in the model then the query will fail even though a simple deductive process that looked in the model for expressions of the form (ON ←X DESK1) could determine an answer to the query. The new design attempts to make it easy to add such deductive processes to the system and thereby turn the retrieval mechanism into a question answering mechanism.

The design allows the user to associate with each predicate in the system (e.g., ON, TOP:CLEAR, IN:ROOM) a list of deductive programs that will be called whenever true instances of an expression containing the predicate are being searched for in the model. Hence, one could associate a deductive program with the predicate TOP:CLEAR that would perform as described in the previous paragraph.

One common use of these deductive programs is to determine that an expression is false based on uniqueness properties of the predicate. For example, if there is a query asking whether OB1 is in room RM1, i.e., (EXISTS (IN:ROOM OB1 RM1)), a deductive program could look in the model for any true expression of the form (IN:ROOM OB1 ←X). If such a true expression is found and the room name that matched with ←X is not RM1, then the deduction can be made that OB1 is not in RM1.

Standard deduction programs are included in the system for NOT, AND, and OR. These programs allow one to ask questions such as: "Find

a desk in either room RM1 or RM2 whose top is clear." The QA4 form of that query might be (EXISTS (AND (TYPE ←X DESK) (OR (IN:ROOM ←X RM1) (IN:ROOM ←X RM2)) (TOP:CLEAR ←X))). To answer this query an AND deductive program is first called. It would ask for an object of type DESK from the model. If one is found, say D1, then it asks if (OR (IN:ROOM D1 RM1) (IN:ROOM D1 RM2)) is true. This query calls an OR deductive program which might in turn call an IN:ROOM deductive program that could determine which room D1 is in by asking the model for the location of D1. If D1 is found to be in either room RM1 or RM2, then the AND program will ask if (TOP:CLEAR D1) is true. This might cause a TOP:CLEAR deductive program to be called that would ask the model for any objects X for which (ON ←X D1) is true; if no such objects are found, then D1 is returned as a desk that satisfies the conditions in the original query.

This question answering process makes use of the QA4 backtracking capabilities when there is a need for more than one answer to a query. For instance, if our example query had been to find all the desks in either room RM1 or RM2 with clear tops, then each time such a desk was found it would be saved in a set and the answering process would be backtracked to a point in the deductive programs where alternative choices were available. In this example, control would return to the search for objects of type DESK, and, if another desk were found, the process would proceed as before to check the room the desk is in and whether its top is clear. Each desk would be checked in this manner to form the desired set.

The backtracking mechanism is also used within the answering process when, for example, a desk that is found by the (TYPE ←X DESK) query is not in either RM1 or RM2. In that case the OR deductive program returns FALSE to the AND deductive program and control is backtracked into the (TYPE ←X DESK) query to find the next desk.

B. A General Tree Searching Package

A set of programs was implemented in QA4 that provides a conceptual framework and executive program for conducting heuristic tree searches. The package assumes that there is an initial node of the tree given and that the task is to find a path through the tree from that initial node to a node that satisfies a given goal test. Whenever the search reaches a node in the tree for the first time, and evaluation number is computed for the node. This evaluation number indicates to the search executive the desirability of the node for further consideration. At each step the executive selects the node with the best evaluation number, generates an offspring node of the selected node, computes an evaluation number for the new node, and tests whether the new node satisfies the goal test. If the goal test fails, then node selection occurs again and the process repeats.

This search package can be applied to a particular task domain by providing the programs for node evaluation, goal testing, and offspring generation. For example, it has been used to compute multiroom paths for a robot system, where each node in the tree represents a room to which the robot has traveled, and each offspring node of a given node represents an adjacent room. Hence, if the initial node represents the room in which the robot begins, then a path through the tree represents a room-to-room route that the robot can travel to reach a goal room. For any applications, including the robot path-finder, we wish the search to find the shortest path from the initial node to the goal. In such cases one often uses a node evaluation scheme that computes for a node an estimate of the length of the shortest path from the initial node to a goal-satisfying node passing through the node being evaluated. This evaluation scheme estimates that the node with the smallest evaluation number is on the shortest path to the goal and therefore should be the one considered by the searcher.

By using QA4 to implement this search package we obtained clear advantages for the user with regard to flexibility, power, ease of use, and search efficiency. Each node in the search tree is represented by a QA4 dynamic context. The search executive proceeds by growing a tree of contexts where the offspring of a given node is formed by doing a context push operation on the node; hence, any variable values and model information that were true in the parent node are automatically true in the offspring node.

The executive assumes that each node has associated with it a node-evaluation function, a goal-test function, and an offspring-generating function. If these functions are to be the same for each node in the tree, as is the case for the robot path-finding example, then they merely need to be attached to the initial node and all other nodes will "inherit" them automatically (via the context mechanism) from their common ancestor, the initial node. But, if the problem domain is such that nodes in different parts of the tree should be evaluated, generated, and tested in different ways, then the nodes can be assigned their individualized functions as they are generated.

The evaluation, generation, and goal testing functions are called by the search executive in the dynamic context of the node being considered; hence, when these functions access model information or variable values, they are given the values that are current at the node. This design frees the writer of these functions from being burdened with concerns about obtaining the correct set of values for the node and handles much of the bookkeeping required in passing information from parent to offspring node.

When a node is selected by the search executive, the node-generation function for that node is used to produce an offspring of the node. The first time any given node is selected by the executive its offspring-

generating function is transformed into a QA4 process. The executive assumes that each time this process is resumed it will form a new offspring and then suspend itself. This implies that the values of local variables in the generating function will be saved between generations so that the function can be written to determine the order in which offspring are generated and thereby significantly affect the amount of searching required to find a solution. For example, the offspring-generating function used in the robot path finder computes the distance from the goal room to an adjacent room before using that adjacent room to create an offspring. If the adjacent room is farther from the goal than the current room, then the room is put on a rejects list and considered again only after the offspring for all the adjacent rooms closer to the goal have been generated. For many problems this computationally inexpensive ordering algorithm virtually eliminates consideration of nodes that are not on a path to a solution.

C. A Hierarchical Robot Planning and Execution System

A robot control program consisting of a hierarchically organized plan generation and execution system was designed and tested with a QA4 implementation. The usually sharp distinction between robot plan generation and execution was intentionally blurred in this system in that planning and execution phases occur intermixed at various levels of the hierarchy. What was desired was a system that could generate plans at various levels of detail and monitor the execution of these plans in a manner well integrated with the planner.

The system models its world with a collection of assertions called WORLD-MODEL. Assertions in the WORLD-MODEL are customarily made or deleted by those actions that actually move the robot in the world, and are not subject to removal by QA4 backtracking. When the system is

generating a plan, it does so with reference to a PLANNING-MODEL. When it is set up, the PLANNING-MODEL is conceptually a copy of the present WORLD-MODEL. However, the QA4 context mechanism allows the PLANNING-MODEL to be formed without the necessity of doing any copying operations. All proposed actions are allowed to have their effect on the PLANNING-MODEL. These effects can be backtracked if an alternative plan needs to be considered.

The fundamental building unit of the hierarchical robot system is the ACTION. Each ACTION is a piece of program, together with its arguments; when it is run it produces some or all of three important effects. The first is simply an effect on the world, such as a robot motion. Secondly, an ACTION may produce or add to a PLAN. A PLAN is an ordered list of ACTIONS. An ACTION may call another ACTION as a subroutine. In such a case the subordinate ACTION may add components to some PLAN that the main ACTION is producing. Thirdly, an ACTION may make changes to various models of the world that the robot system maintains. If the ACTION has an effect on the world, it records this effect in the WORLD-MODEL. If the ACTION produces a PLAN, the predicted effects of running this plan are recorded in a PLANNING-MODEL that has been specially set up before running the ACTION routine.

All planning and executions are diffused throughout the entire system in such ACTION programs; there is no special top-level planning program that produces plans to be executed by a top level executive. Through the use of the QA4 GOAL statement, planning that requires "search" can also be done by the system. QA4 takes care of such search processes automatically through its backtracking mechanism.

EXECUTIVEs are the programs that execute PLANs. Each EXECUTIVE is run as a QA4 process that creates offspring EXECUTIVE processes. At any given time, there may be several EXECUTIVE processes set up and

waiting to run. The argument of an EXECUTIVE program is a PLAN (i.e., an ordered list of ACTIONs). The EXECUTIVE program can be described simply as an ordered application of a program called PERFORM to each ACTION in the PLAN. Figure 1 shows how PERFORM works.

The system uses QA4 demons to deal with the happy kind of surprise that signals that some goal is satisfied before the robot expected it would be. In this case, the system should abandon all work toward making that goal true and get on with the rest of the task. A QA4 demon specifies a "watch-for" condition and a program to be run whenever the condition being watched is met. This system stores on the property list of a PLAN that particular assertion that the PLAN is supposed to achieve. Then, just before an EXECUTIVE process is created to run the PLAN, a demon is set up to watch the WORLD-MODEL for the occurrence of the assertion. Whenever this assertion occurs, either as a direct result of the robot's efforts or through serendipity, the demon interrupts whichever process is running and transfers control to the EXECUTIVE process that originally set up the demon. In this way, the next step in the next-higher level PLAN will be executed. This program makes use of many of the features of QA4; in particular, it relies heavily on pattern directed function invocation, automatic backtracking, processes, retrieval and storage of expressions in the QA4 net, associational property lists with expressions, and demons.

D. Program Verification

One major application of QA4 has been program verification. A theorem prover has been constructed to prove the verification conditions necessary to certify the correctness of programs.[13,14] The QA4 language was found to be quite well suited for this domain. The theorem prover is concise, readable, and easy to extend and modify. It embodies a good

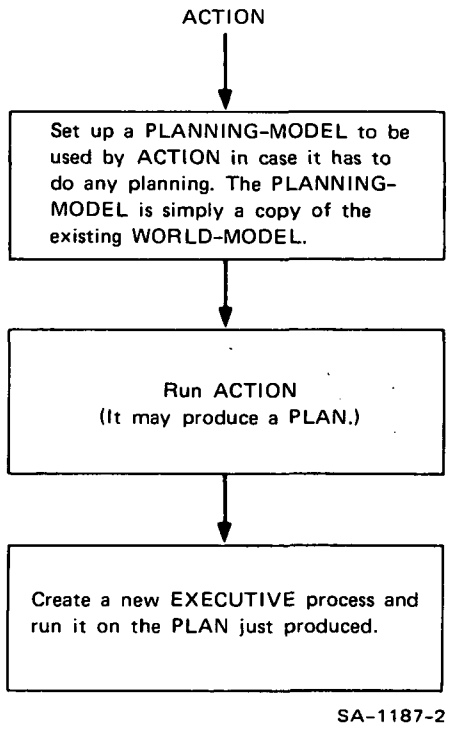

FIGURE 1 SIMPLIFIED FLOW CHART FOR PERFORM

deal of knowledge about numbers, equality, ordering relations, arrays, lists, and other programming concepts. This information is expressed by more than one hundred small QA4 functions. Pattern directed function invocation, and in particular the goal mechanism, is used to call the functions appropriate to solve a given problem. We will look at some of these functions to see why QA4 is an appropriate vehicle for this application.

One function, EQSIMP, represents the tactic that "to prove an expression of the form X=Y, try to simplify X, and then prove that the simplified X is equal to Y:"

```
EQSIMP =
  (LAMBDA (EQ ←X ←Y)
          (PROG ()
                (SETQ ←X ($SIMPONE $X)
                (GOAL $EQRULES (EQ $X $Y)))
          (BACKTRACK) ).
```

The bound variable part, (EQ ←X ←Y), expresses what sort of goal this tactic is relevant to. SIMPONE is the name of the simplifier, and EQRULES is the class of functions, including EQSIMP itself, applicable to proving equalities.

Notice that this one rule can be used to prove the right side of an equality as well as the left; the predicate EQ has been declared to take a set as its argument. Thus (EQ ←X ←Y) is an abbreviation for (EQ (SET ←X ←Y)). Furthermore, EQSIMP specifies the BACKTRACK option. When applying EQSIMP to an expression of form (EQ A B) (i.e. (EQ (SET A B))), the pattern matcher will chose an arbitrary match, of (EQ (SET ←X ←Y)) to (EQ (SET A B)), binding X to A and Y to B, for instance. If SIMPONE should be unable to simplify A, it will fail; the system will backtrack to the point at which the match was chosen, to B, and will select the

alternative binding, X to B and Y to A. EQSIMP will be executed again with the new binding, simplifying B instead of A. Thus, the backtracking mechanism combined with the set data structure allows us an unusually concise representation of this sort of rule.

A common operation in theorem proving is simplification. In the theorem prover, each simplification rule is represented as a separate QA4 function. When an expression is to be simplified, the appropriate rules are summoned by means of the pattern-directed function invocation mechanism. One such rule expresses the fact that X + 0 = +X:

```
PLUSZERO =
  (LAMBDA (PLUS ←←X 0)
          ('(PLUS $$X))) .
```

Note that doubly-prefixed variables represent a fragment, a sequence of elements rather than a single element, and that PLUS has been declared to take a "bag" as its argument.

Thus (PLUS X 0) is an abbreviation for (PLUS (BAG X 0)), and this rule can simplify

0 + A into +A

or even

A + B + 0 + C + D + E into A + B + C + D + E

The use of bags combined with fragments provides an unusual conciseness here.

The QA4 special relations handler was especially applicable to the program verification domain. This system stores and accesses information about equality and inequality in a way consistent with the particular properties of these relations, such as transitivity and symmetry, asymmetry or antisymmetry. For instance, if one asserts

$$A \leq B \text{ and}$$

$$B \leq C,$$

the system will know immediately that

$$A \leq C.$$

If one asserts

$$A \leq B \text{ and}$$

$$A \neq B,$$

the system will know

$$A < B.$$

If one asserts

$$A \leq B \leq C \leq D \leq A ,$$

the system will be able to ascertain the functional relationship

$$(F\ U\ B\ V) = (F\ U\ C\ V)$$

This sort of processing is built into the system and need not be represented by QA4 functions.

Although most of the deductions performed by the theorem prover proceed backward from a goal, some reasoning progresses forward from assertions. This reasoning is carried out by demons. These demons are triggered off by assertions and denials. For instance, one demon, whenever an assertion of the form $X + Y \leq X + Z$ is made, will assert $Y \leq Z$. The formal specification for this demon is:

```
(WHEN EXP (LTQ (PLUS ←X ←Y)
               (PLUS ←X ←Z))
     INDICATOR MODELVALUE
     THEN (ASSERT (LTQ $Y $Z)))
```

Another demon--applicable only to the integer domain--will conclude, whenever an expression of the form $X < Y$ is asserted, that $X + 1 \leq Y$:

```
(WHEN EXP (LT ←X ←Y)
      INDICATOR MODELVALUE
      THEN (ASSERT (LTQ (PLUS $X 1) $Y))).
```

For example, these two demons, working with the special relations handler, will, after the user has asserted

$$A \leq B$$

and

$$B < A + 1,$$

enable the system to conclude that

$$A = B.$$

The modular design of the theorem prover makes it easy to incorporate information about new subject domains: new functions may be defined by an educated user, the new functions will be invoked by means of the goal mechanism when appropriate, and the addition of new information does not noticeably degrade the behavior of the system when it is applied to problems that do not need that information.

The theorem prover has been translated from QA4 into QLISP with appreciable improvement in running speed.

It is expected that future work on program verification and automatic programming at SRI will be done in QLISP. QLISP is being used in a current project at SRI on interactive program synthesis. Work on automatic programming directed by Professor Cordell Green at Stanford uses QLISP as the implementation language. Two programs related to this project now exist in QLISP, one generating programs from formal specifications and the other generating programs from sample inputs and outputs.

REFERENCES

1. J. F. Rulifson, J. A. Derksen, and R. J. Waldinger, "QA4: A Procedural Calculus for Intuitive Reasoning," Technical Report, Contract NASW-2086, SRI Project 8721, Stanford Research Institute, Menlo Park, California (November 1972).

2. R. A. Yates, B. Raphael, and T. P. Hart, "Resolution Graphs," Artificial Intelligence, Vol. 1, No. 4, pp. 257-289 (1970).

3. C. C. Green and B. Raphael, "The Use of Theorem-Proving Techniques in Question-Answering Systems," Proc. 23rd National Conf., ACM, pp. 169-181 (1968).

4. C. C. Green, "Application of Theorem Proving to Problem Solving," Proc. International Joint Conference on Artificial Intelligence, pp. 219-239 (The MITRE Corp., Bedford, Mass., 1969).

5. B. Raphael, "Programming a Robot," Proc. IFIP Congress 68, Vol. 2, pp. 1575-1581 (1969).

6. T. D. Garvey and R. E. Kling, "User's Guide to QA3.5 Question-Answering System," Technical Note 15, Artificial Intelligence Group, Stanford Research Institute, Menlo Park, California (December 1969).

7. D. G. Bobrow and B. Raphael, "New Programming Languages for AI Research," Technical Note No. 82, Artificial Intelligence Center, Stanford Research Institute, Menlo Park, California (August 1973).

8. D. Swinehart and B. Sproull, "SAIL," Operating Note No. 57.2, Stanford Artificial Intelligence Project, Stanford University, Stanford, California (January 1971).

9. C. Hewitt, "PLANNER: A Language for Proving Theorems in Robots," Proc. IJCAI, pp. 295-301 (The MITRE Corp., Bedford, Mass., 1969).

10. D. V. McDermott and G. J. Sussman, "The CONNIVER Reference Manual," MIT AI Memo No. 259, Massachusetts Institute of Technology, Cambridge, Massachusetts (May 1972).

11. D. Davies and M. Julian, "POPLER 1-5 Reference Manual," TPU Report No. 1, University of Edinburgh, Edinburgh, Scotland (May 1973).

12. J. F. Rulifson, "Research in Advanced Formal Theorem-Proving Techniques," Final Report, Contract NASW-2086, SRI Project 8721, Stanford Research Institute, Menlo Park, California (June 1971).

13. B. Elspas, K. N. Levitt, and R. J. Waldinger, "An Interactive System for the Verification of Computer Programs," Final Report, SRI Project 1891, Stanford Research Institute, Menlo Park, California (in preparation).

14. R. J. Waldinger and K. N. Levitt, "Reasoning about Programs," Technical Note, Artificial Intelligence Center, Stanford Research Institute, Menlo Park, California (in preparation).

Appendix

A PRELIMINARY QLISP MANUAL

Page Intentionally Left Blank

ABSTRACT

A preliminary version of QLISP is described. QLISP permits free intermingling of QA4-like constructs with INTERLISP code. The preliminary version contains features similar to those of QA4 except for the backtracking of control environments. It provides several new features as well.

This preliminary manual presumes a familiarity with both INTERLISP and the basic concepts of QA4. It is intended to update rather than replace the existing documentation of QA4.

I INTRODUCTION

QLISP brings together in a natural fashion the control structures, pattern matching, and net storage mechanisms of QA4[*1], and the versatility, programming ease, and interactive features of INTERLISP[2] (formerly called BBN-LISP). The system permits free intermixing of QA4, CLISP[3], and the INTERLISP code so that, for example, users may write programs or type expressions for evaluation in LISP, CLISP, or QA4, or in a mixture of all three.

QLISP has been implemented by means of the error correction facility in INTERLISP. A valid LISP expression will never be seen by the QLISP processor. Thus, programs that do not use QA4 constructs will run as fast in QLISP as in LISP. When the LISP interpreter encounters an ill-formed expression, it calls an error routine that in turn invokes the error analyzer. If the form is recognized as a QA4 construct, it is translated to an equivalent LISP form that is returned to the interpreter for evaluation; if the expression is a valid CLISP construct, a similar translation takes place.

In either case, the translation is stored with the original QLISP expression so that the analysis and translation are done only once.

Since QLISP allows QA4 operations to be embedded directly in LISP, the new system does not require the QA4 interpreter or stack mechanism. Furthermore, QLISP programs are stored as standard LISP functions rather than as expressions in the QA4 discrimination net, thus reducing the search time required for associative retrievals as well as for function calls.

---

* Superscripts denote references listed at the end of the report.

A preliminary version of QLISP is now available. It provides features similar to those of QA4 except for the backtracking of control environments. It provides several new features as well.

The major differences between this version of QLISP and the current QA4 are:

- QLISP functions are defined and stored in the same way as LISP functions. Functions are stored as values of net variables in QA4.
- The QA4 data-type SET has been renamed CLASS, and a new data-type VECTOR has been added.
- The QA4 SETQ and SETQQ statements have been renamed MATCHQ and MATCHQQ. In addition, QLISP provides a MATCH statement (the pattern-matching equivalent of the LISP function SET).
- The QA4 statement EXISTS has been renamed IS, and a corresponding statement ISNT has been added.
- The syntax of the net storage and retrieval statements has been modified.
- "Demons" have been replaced by "teams" of functions that may be applied by any net storage or retrieval function.
- Tuples are now equivalent to LISP lists. The external form of (TUPLE a_1 a_2 ... a_n) is now (a_1 a_2 ... a_n) rather than (TUPLE a_1 a_2 ... a_n). The internal form of (a_1 a_2 ... a_n) is now (TUPLE a_1 a_2 ... a_n) rather than (APPL a_1 (TUPLE a_2 a_3 ... a_n)).
- Processes are not currently available.
- Backtracking out of the scope of a single statement, e.g., backtracking to a previous IS, MATCH, or GOAL statement is not currently possible. However, as a temporary expedient, the effect of backtracking to an IS can be simulated by using BIS ("Backtrack IS").

This is a preliminary QLISP manual. It is intended to update rather than to replace the existing documentation of QA4. It presumes a familiarity with both INTERLISP and QA4.

We are convinced that QLISP can be the most usable of the recent generation of AI languages. With this in mind, the authors hereby elicit and encourage feedback from the user community.

A sample QLISP program is given in Annex A. The syntax of QLISP statements is given in Annex B.

II THE QLISP LANGUAGE

A. QLISP Expressions

QLISP provides complete freedom in intermingling LISP expressions with those that provide net storage and retrieval, pattern matching, and control structure manipulation. As an example, consider the ARE-COUSINS program:

```
(QLAMBDA (←PERSON1 ←PERSON2)
        (IS (FATHER $PERSON1 ← F))
        (IS (UNCLES $PERSON2 ←← U))
        (IF (MEMB $F $U)
            THEN (PRINT (' ($PERSON1 AND $PERSON2 ARE COUSINS)))
            ELSE (PRINT (' ($PERSON1 AND $PERSON2 ARE NOT COUSINS)))))
```

When this program is executed, two associative retrievals from the discrimination net will obtain the father of the first person and the uncles of the second person. If the father of the first person is among the uncles of the second, we proclaim the two persons to be cousins.

B. The QLISP Discrimination Net

QLISP provides data base storage and retrieval facilities similar to those of QA4. Data of type TUPLE, VECTOR, BAG, or CLASS may be stored in a discrimination net and accessed by associative retrieval.

A tuple is equivalent to a LISP list. The form $(f\ a_1\ a_2\ \ldots\ a_n)$ will be stored internally as $(\text{TUPLE}\ f\ a_1\ a_2\ \ldots\ a_n)$ unless f is declared to take a VECTOR, BAG, or CLASS as its argument. Such a declaration is performed by evaluating DEFTYPE[f;type], where f is a function or predicate name and type is VECTOR, BAG, or CLASS.

A vector is similar to a tuple, that is, it contains an ordered

sequence of elements. The treatment of tuples and vectors differs only if they are evaluated. The value of a tuple is the result of applying the function specified by its first element to the values of the rest of its elements; the value of a vector is simply a vector of the values of its elements. Thus a tuple is useful for representing an evaluable form containing a function and its arguments, whereas a vector is useful for representing an argument list alone.

A bag is a collection of unordered elements, and the elements may be duplicated, that is (BAG A A B C) is EQ to (BAG A C B A).

A class is a collection of unordered elements, without duplication, that is, (CLASS A B C A) is EQ to (CLASS C B A).

C. Constructing QLISP Expressions

The QLISP functions TUPLE, VECTOR, BAG, and CLASS cause an expression of the appropriate type to be built and dropped into the discrimination net. For example, (BAG F A B) will create a bag with elements A, B, and F. Similarly, (TUPLE F A B) will create the tuple (F A B) regardless of whether F was DEFTYPEd.

The construction of new expressions is simplified by the use of the STRIP operator(!). The STRIP operator causes a level of parentheses to be stripped from its argument (and thus is meaningless at the top level).

For example, (VECTOR (!(VECTOR A B))(!(VECTOR B C))) will cause the vector (VECTOR A B B C) to be built.

Note that (! $X) is equivalent to $$X.

D. Inverse Quote Mode, Evaluation, and Instantiation

Statements for net storage and retrieval, pattern matching, and control structure manipulation are interpreted in inverse quote mode. That

is, atoms that are not otherwise identified are treated as constants. Variables are indicated by a prefix character or characters.

These statements normally instantiate their arguments rather than evaluate them. To instantiate an expression is simply to replace its net variables by their values. For example, if the variable $X is bound to B, then the QLISP statement (ASSERT (FOO A $X)) will assert the expression (FOO A B), not the result of evaluating (FOO A B).

The "at" sign (@) is used to force evaluations of the following expression. For example, (ASSERT (@ (FOO A $X))) will assert the result of evaluating (FOO A B).

The quote mark(') is used to force instantiation where evaluation would normally take place. For example, (PRINT ('(THIS EXPRESSION WILL $X INSTANTIATED))) will cause (THIS EXPRESSION WILL B INSTANTIATED) to be printed.

The colon prefix (:) is used to prevent the instantiation of a $ variable when it occurs in an expression to be instantiated. For example, (PRINT ('(THE VALUE OF :$X IS $X))) will cause (THE VALUE OF $X IS B) to be printed.

E. QLISP Functions

QLISP functions are of three varieties: LAMBDA and NLAMBDA, as in INTERLISP, and QLAMBDA, which is equivalent to the QA4 LAMBDA and has a pattern as its bound variable part. QLAMBDA expressions admit "implicit PROGNs" just as LAMBDAs and NLAMBDAs do.

F. Declaring Local Variables

The QPROG statement is an analog of the LISP PROG feature. It allows the declaration of net variables in the local context as well as

local LISP variables. The format of the QPROG statement is:

(QPROG args e_1 ... e_n).

LISP variables are denoted in the args list exactly as in the LISP PROG statement. Net variables are denoted by prefixing them with a left arrow (←).

The expressions e_i are arbitrary QLISP expressions.

For example, the statement

```
(QPROG (U←V (W Ø) (←X (TUPLE)))
       .
       .
       .
       )
```

will cause local variables U,V,W, and X to be declared. U and W will be LISP variables. V and X will be net variables. U will have an initial value of NIL, V's initial value will be NOSUCHPROPERTY W will be bound to Ø, and X will be bound to the empty tuple.

Both PROG and QPROG should be exited by either of the functions RETURN or QRETURN. QRETURN is similar to the LISP RETURN, except that it instantiates its argument rather than evaluating it.

G. QLISP Backtracking

The side effects of QLISP computations may be undone (in the INTERLISP sense) by the use of the QLISP failure mechanism. A statement that invokes pattern matching will fail if no match exists or if all matches have been exhausted. Other statements may be caused to fail by the use of the FAIL statement which is described below.

A failure will cause a return to some backtrack point and undo all undoable computations performed since the backtrack point was established.

Manipulation of expressions in the net is undoable unless it is

done with respect to the context ETERNAL, or one of its descendants. Manipulation of list structures is undoable if it is done by means of "/ functions."[4] In addition to the functions provided by INTERLISP, QLISP has a /SETQ function as well.

Backtrack points are established within all net storage and retrieval statements and within QLAMBDA expressions that have the BACKTRACK option.

Failures may be caused explicitly by executing the FAIL statement. Its format is (FAIL name).

If name is absent or NIL, FAIL causes a failure.

If name is CALLER, FAIL causes the last net storage or retrieval statement to fail.

If name matches the NAME of a net storage or retrieval statement, FAIL causes the named statement to fail. (Assignments of NAMEs to statements and examples of the use of the FAIL statement will be discussed in the following section.)

H. Statements of the QLISP Language

A full listing of the syntactic form of each non-LISP statement appears in Annex B. New statements and those that differ significantly from their old QA4 counterparts are discussed below.

In the syntactic forms to follow, braces ({}) indicate an optional clause. The ordering of options is arbitrary.

The net storage and retrieval statements have a new common syntax. First, the syntax will be discussed in general, and then each of these statements will be described.

1. Syntax of Net Storage and Retrieval Statements

The general form of a QLSIP net storage and retrieval statement is:

(statement-type p-exp {APPLY team} {WRT ctx} {NAME name}
{ind_1 $prop_1$ ... ind_n $prop_n$})

The statement-type is one of the following:

QPUT
QGET
ASSERT
DENY
IS
ISNT
BIS
INSTANCES
GOAL
CASES
DELETE

p-exp is a pattern to be instantiated.

The team must instantiate to a class, a bag, or a tuple of QLAMBDA functions. They will be applied to the instantiation of p-exp. If a failure occurs during the application of one function in the team, its side effects are undone and the next function in the team is tried. With the exception of the GOAL statement, the application of the team functions is for their effect rather than their values (i.e., the values returned by the team functions are never used by the calling statement).

ctx, when present, must instantiate to a context. This context will be passed as a context recommendation to the functions in the team (i.e., it will be used as a default value for context references in the team functions). The following are the possible context specifications and their meanings:

- LOCAL--Current context.
- GLOBAL--Top context.
- ETERNAL--Current default context. Changes made in this context will not be backtrackable.
- UNIVERSAL--Top context. Changes made in this context will

not be backtrackable.

- A variable that was previously bound to a context.
- A CONTEXT statement.

If the WRT option is omitted in the statement, the context is bound by default to the last context recommendation from a calling statement. If no such recommendation was made, the top context is used.

The statement may be given a name. A team function may refer to that name in a FAIL statement. The default name is the type of the statement itself (e.g., ASSERT).

ind_1 $prop_1$ ... ind_n $prop_n$ are property specifications. Their use in each statement type is discussed below.

2. Net Storage and Retrieval Statements

a. QPUT

The format of the QPUT statement is:

(QPUT p-exp {APPLY team} {WRT ctx} {NAME name}
{ind_1 $prop_1$ ... ind_n $prop_n$})

The steps in the evaluation of a QPUT statement are as follows:

(1) The pattern p-exp is instantiated, and an expression exp that matches the instantiation is retrieved from the net.

(2) A context CTX in which to evaluate the statement is determined as described above.

(3) The indicator-property pairs are instantiated, and all properties $prop_i$ are assigned to the expression under the corresponding indicators ind_i with respect to CTX.

(4) If the NAME option is present, then the statement is named name. Otherwise the statement is named QPUT.

(5) If there is an APPLY option, team is instantiated and all of its functions are applied successively to

exp. The default for context references in the team functions will be CTX.
If any team function fails, its side effects are undone.
The functions in the team may cause the QPUT statement itself to fail, as described in Section II-G, above.

(6) The statement returns exp as its value. For example, in the sample program in Annex A. the QPUT statement in HITCH,
(QPUT(PERSON $HUMAN)MARRIEDTO $Y APPLY $COMPUTERELATIONS)
operates as follows when the argument to HITCH is (HAPPY ADAM)

- The pattern (PERSON $HUMAN) is instantiated to (PERSON ADAM), and (PERSON ADAM) is retrieved from the net.
- The statement will be evaluated in the top context, because no context was specified and the current default context is the top context.
- $Y is instantiated to ADAMs spouse, say, EVE, and placed as the value of the indicator MARRIEDTO on the property list of (PERSON ADAM).
- The statement is named "QPUT."
- The team $COMPUTERELATIONS is instantiated to the tuple (MAKESPOUSE).
  The function MAKESPOUSE is called with the argument (PERSON ADAM); if the (FAIL CALLER) statement in MAKESPOUSE is executed, the QPUT statement itself will fail and the property list of (PERSON ADAM) will be restored.
  If a (FAIL) statement instead of (FAIL CALLER) had been executed, then the side effects of MAKESPOUSE would have been undone but execution of the QPUT statement would have continued.
- (PERSON ADAM) is returned as the value of the QPUT statement.

b. QGET

The format of the QGET statement is:

(QGET p-exp {APPLY team} {WRT ctx} {NAME name}

$$\left[\begin{array}{l} \text{ind} \\ \text{ind}_1\ \text{prop}_1\ \{\text{ind}_2\ \text{prop}_2\ \ldots\ \text{ind}_n\ \text{prop}_n\} \end{array}\right])$$

The evaluation of a QGET statement is similar to that of QPUT, except that it retrieves values for each indicator ind_i. If no property can be found for an indicator, NOSUCHPROPERTY is used as its value. The values are matched against the corresponding patterns prop_i. If a match for p-exp cannot be found, or if the match to some prop_i does not succeed, the QGET statement fails.

If only one indicator (and no properties) was specified, QGET returns the corresponding property value. Otherwise, QGET returns the expression exp.

For example, in the sample program in Annex A, when the QGET statement in CHECKAGE,

```
(QGET (PERSON (@ SPOUSE)) SEX ←SEX AGE ←AGE)
```

is called with EVE as the value of SPOUSE, then $SEX will be bound to her sex, $AGE will be bound to her age, and (PERSON EVE) will be returned as the value of the QGET statement.

c. ASSERT

The format of the ASSERT statement is:

(ASSERT p-exp {APPLY team} {WRT ctx} {NAME name}
{ind_1 prop_1 ... ind_n prop_n})

ASSERT performs (QPUT p-exp {APPLY team} {WRT ctx} {NAME name}
MODELVALUE T {ind_1 prop_1 ... ind_n prop_n}).

Responsibility for consistency checks rests with the functions in the APPLY team.

d. DENY

The format of the DENY statement is:

(DENY p-exp {APPLY team} {WRT ctx} {NAME name}
{ind_1 $prop_1$ ... ind_n $prop_n$})

DENY performs (QPUT p-exp {APPLY team} {WRT ctx} {NAME name}
MODELVALUE NIL {ind_1 $prop_1$ ... ind_n $prop_n$})

Responsibility for consistency checks rests with the functions of the APPLY team.

e. IS

The format of the IS statement is:

(IS p-exp {APPLY team} {WRT ctx} {NAME name}
{ind_1 $prop_1$ ... ind_n $prop_n$})

IS performs (QGET p-exp {APPLY team} {WRT ctx} {NAME name}
MODELVALUE T {ind_1 $prop_1$ ... ind_n $prop_n$}).

f. ISNT

The format of the ISNT statement is:

(ISNT p-exp {APPLY team} {WRT ctx} {NAME name}
{ind_1 $prop_1$ ... ind_n $prop_n$})

ISNT performs (QGET p-exp {APPLY team} {WRT ctx} {NAME name}
MODELVALUE NIL {ind_1 $prop_1$ ... ind_n $prop_n$})

g. BIS

The format of the BIS statement is:

(BIS p-exp {APPLY team} {WRT ctx} {NAME name}
{ind_1 $prop_1$ ... ind_n $prop_n$} THEN e_1 {e_2 ... e_m})

BIS performs the same retrieval as IS, but when an expression has been found and the team members applied, a backtrack point is established and the expressions $e_1 \ldots e_m$ are evaluated in turn. If a failure occurs, BIS will continue to retrieve different expressions from the net until either none of the e_i fails, in which case BIS returns the retrieved expression, or until all possible retrievals from the net have been attempted, in which case BIS fails.

BIS is a temporary expedient, which may be removed from the language when INTERLISP permits control structure backtracking. At that time the IS statement will be able to establish a backtrack point, and failures below it will cause another attempt at retrieval to take place.

For example, in the sample program of Annex A, when HITCH is called with the argument (HAPPY ADAM), the BIS statement will retrieve an expression from the net of the form (PERSON ←Y) that has the property FEMALE under the indicator SEX. The statements after THEN will be evaluated. If any statement fails, another expression will be retrieved from the net, and the cycle will be repeated. If no statement fails BIS will return (HAPPY ADAM). If no expression had been found in the net for which none of the statements failed, then the BIS statement would have failed.

h. INSTANCES

The format of the INSTANCES statement is:

(INSTANCES p-exp {APPLY team} {WRT ctx} {NAME name}
{ind_1 $prop_1$ ... ind_n $prop_n$})

INSTANCES instantiates the pattern p-exp, determines a context CTX, and computes a name as was described for the QPUT statement.

INSTANCES then retrieves all the expressions from the net that match the instantiation of p-exp, that have the value

indicator MODELVALUE (unless some ind_i is MODELVALUE) and that have properties on the indicators ind_i which match the property patterns $prop_i$, with respect to the context CTX.

For each such expression found, all the members of the APPLY team are applied to it successively.

A CLASS of all the retrieved expressions is returned as the value of the statement.

i. GOAL

The format of the GOAL statement is:

(GOAL p-exp {APPLY team} {WRT ctx} {NAME name}
{ind_1 $prop_1$ ... ind_n $prop_n$}).

GOAL first performs (IS p-exp {WRT ctx} {NAME name}
{ind_1 $prop_1$ ... ind_n $prop_n$}).

If an expression was found, it is returned as the value of the statement. If not, the functions of the team are applied successively to the instantiated p-exp until some team member does not fail. The value returned by that team member is then returned as the value of the statement. If all the functions of the team fail, the GOAL statement fails.

j. CASES

The format of the CASES statement is:

(CASES p-exp {APPLY team} {WRT ctx} {NAME name}).

CASES is equivalent to GOAL, except that the IS statement is not performed.

k. DELETE

The format of the DELETE statement is:

(DELETE p-exp {APPLY team} {WRT ctx} {NAME name}
{ind_1 $prop_1$ ... ind_n $prop_n$})

DELETE performs the equivalent of

(INSTANCES p-exp {APPLY team} {WRT ctx} {NAME name}
MODELVALUE (POR T NIL) {ind_1 $prop_1$ ... ind_n $prop_n$})

For each expression of the CLASS returned by INSTANCES, the indicator MODELVALUE is removed. DELETE returns that CALSS.

3. Other Statements That Can Set a Default Context

a. DO

The format of the DO statement is:

(DO tpl WRT ctx)

DO instantiates tpl in the current context. The pattern tpl must instantiate to a pattern of SYPTE TUPLE. The WRT clause is evaluated, and this determines a new default context. The instantiation of tpl is evaluated in this context, and the result of the evaluation is returned as the value of DO.

b. MATCH

The format of the MATCH statement is:

(MATCH exp val {WRT ctx})

MATCH evaluates exp and val. It then attempts to match the value of val to the value of exp. Any new variable bindings that are created by the matching process are made with respect to ctx, if present, or to the current context. The value of MATCH is the value of val.

c. MATCHQ

MATCHQ is identical to MATCH except that the argument exp is instantiated rather than evaluated.

This statement corresponds to the SETQ statement in QA4.

d. MATCHQQ

MATCHQQ is identical to MATCH except that both exp and val are instantiated rather than evaluated.

This statement corresponds to the SETQQ statement in QA4.

III HOW TO USE QLISP

A. Loading the System

The QLISP system may be loaded into LISP by typing:

SYSIN(⟨SACERDOTI⟩QLISP.SYS) .

All subsequent type-ins will be processed by the full QLISP system.

B. Creating and Using Symbolic Files

QLISP uses the INTERLISP file package to manipulate symbolic files. PRETTYDEF, MAKEFILE, LOAD, and LOADFNS all know about QLISP constructs and handle them properly. Variables whose values are to be extracted from the net on output and stored into the net on input should be prefixed by a $. Input to and output from the net is done with respect to the current dynamic context at the time of the I/O operation. Input of net variables is done with respect to the context ETERNAL, and thus the values of LOADed variables are not backtrackable (although the effect of a LOAD is UNDOable if it is initiated at the teletype).

C. Defining Functions

QLISP functions are defined by using the functions PUTD, DEFINE, and DEFINEQ.

D. Editing Functions and Variables

All QLISP functions may be edited by using EDITF.

QLISP variables may be edited by using EDITV. Net variables should be prefixed by a $. Retrieval from and storage into the net is done

with respect to the current dynamic context at the time EDITV is called. If the value of a net variable that is not at the top context is being edited, EDITV will print a warning message.

E. Tracing QLISP Functions

Since the current implementation of LISP does not permit us to backtrack properly through BREAKS, we have implemented a trace facility that ADVISEs rather than BREAKs functions.

Functions may be traced by executing the function QTRACE. It is an NLAMBDA no-spread function (just like TRACE), and thus can accept a single function name or a sequence of function names.

The tracing of functions may be turned off by invoking the function UNQTRACE, which is also an NLAMBDA no-spread. Calling UNQTRACE with no arguments causes all traced functions to be untraced.

When new functions are defined by loading a symbolic file or by explicit calls to PUTD, DEFINE or DEFINEQ, all QLAMBDA functions will be automatically QTRACEd if the global variable QTRACEALL is set to T. (It is set to T when QLISP is first loaded.)

QTRACE output may be directed to a file other than the teletype by executing the command TRFILE (file name). This will also set LINELENGTH to 120 to conserve line printer paper. The command UNTRFILE () will redirect QTRACE output back to the teletype, reset LINELENGTH, and close the previous output file.

F. Restrictions and Caveats

All function, variable, and property names beginning with "QA4:" are reserved for the QLISP system.

LISP variables should not begin with $, ?, or ←.

With one exception, all CLISP constructs are valid in QLISP. The exception is the use of the quote mark ('), which is used in QLISP to denote the quasi-quote rather than the LISP QUOTE.

ACKNOWLEDGMENTS

In creating QLISP, we have assembled in one package the good ideas of many individuals. We are particularly indebted to Richard Waldinger, Richard Fikes, Jeff Rulifson, Warren Teitelman, and Mark Stickel. The development of the QLISP language was supported by the National Aeronautics and Space Administration under Contract NASW-2086. This work was made possible by the environment and facilities of the SRI Artificial Intelligence Center, which has been largely supported by the Advanced Research Projects Agency through Contract DAHC04-72-C-0008.

REFERENCES

1. J. Rulifson, Derksen, J. A., and Waldinger, R. W., "QA4: A Procedural Calculus for Intuitive Reasoning," AIC Technical Note 73, Stanford Research Institute, Menlo Park, California (November 1972).

2. W. Teitelman, et al., BBN-LISP TENEX Reference Manual, Bolt Beranek and Newman, Cambridge, Massachusetts (July 1971).

3. W. Teitelman, "CLISP--Conversational LISP," Third International Joint Conference on Artificial Intelligence, Advance Papers of the Conference, pp. 686-690, Stanford Research Institute, Menlo Park, California, August 1973).

4. Teitelman, W., et al., op. cit.

Page Intentionally Left Blank

Annex A

A SAMPLE QLISP PROGRAM

Page Intentionally Left Blank

Annex A

A SAMPLE QLISP PROGRAM

With the goal of creating a better world through computer science, we present below a small system for making people happy. It was written not with elegance or efficiency in mind, but to give examples of the new QLISP features and their interactions with INTERLISP.

A. Program Testing

This symbolic file was created by MAKEFILE in the ordinary LISP fashion.

```
; <REBOH>GENESIS.;3    MON 13-AUG-73 2:25PM

(FILECREATED "19-JUL-73 16:44:07" GENESIS)

(DEFINEQ

(SETUP
  [LAMBDA NIL                                          (* INITIALIZATION
                                                       ROUTINE.)
    (ASSERT (PERSON MARY)
            SEX FEMALE AGE 30 HOBBIES (CLASS TENNIS NEEDLEPOINT DANCING)
            )
    (ASSERT (PERSON ALICE)
            SEX FEMALE AGE 72 HOBBIES (CLASS SCUBA-DIVING BIRD-WATCHING)
            )
    (ASSERT (PERSON EVE)
            SEX FEMALE AGE 29 HOBBIES (CLASS SNAKE-CHARMING GARDENING
                                             VOLLEYBALL))
    (ASSERT (PERSON ADAM)
            SEX MALE AGE 30 NETWORTH 500000 HOBBIES
            (CLASS HUNTING FISHING GARDENING))
    (ASSERT (PERSON SARA)
            SEX FEMALE AGE 40 NETWORTH 2000000])

(MAKEHAPPY
  [LAMBDA (L)                                          (* 'L IS A LIST OF
                                                       PERSONS.)
                                                       (* TRY TO MAKE EACH
                                                       PERSON HAPPY.)
    (MAPC L (FUNCTION (LAMBDA (X)
              (PRINT (ATTEMPT (GOAL (HAPPY (@ X))
                                    APPLY
                                    (TUPLE HITCH RICH]
    (QUOTE FINISHED]))

(HITCH
  [QLAMBDA (HAPPY ←HUMAN)                              (* CYCLE THROUGH ALL
                                                       MEMBERS OF THE OPPOSITE
                                                       SEX.)
                                                       (* HYPOTHESIZE A
                                                       MARRIAGE AND SEE IF IT
                                                       WORKS OUT.)
                                                       (* IF IT DOES, THEN THE
                                                       HUMAN IS HAPPY.)
           (BIS (PERSON ←Y)
                SEX
                [@ (PARTNERSEX ('(PERSON $HUMAN]
                THEN (ASSERT (MARRIED $HUMAN ←Y)
                             APPLY $MARRIAGEDEMONS)
                     (QPUT (PERSON $HUMAN)
                           MARRIEDTO $Y WRT GLOBAL APPLY
                           $COMPUTERELATIONS)
                     ('(HAPPY $HUMAN])
```

```
1  <REBOH>GENESIS.;3    MON 13-AUG-73 2:25PM

(CHECKAGE
  (QLAMBDA (MARRIED *COUPLE)                          (* MAKE SURE THE WIFE IS
                                                      NOT TOO MUCH OLDER THAN
                                                      THE HUSBAND.)
           (QPROG (*SEX
                   *AGE
                   MALEAGE FEMALEAGE)
                  (MAPC (CDR $COUPLE)
                        (FUNCTION (LAMBDA (SPOUSE)
                            (QGET (PERSON (@ SPOUSE))
                                  SEX *SEX
                                  AGE *AGE)
                            (IF (EQ $SEX (QUOTE MALE))
                                THEN (SETQ MALEAGE $AGE)
                                ELSE (SETQ FEMALEAGE $AGE]
                  (IF (GREATERP FEMALEAGE (PLUS MALEAGE 5))
                      THEN (FAIL CALLER))
                  (QRETURN OK))))

(CHECKHOBBY
  (QLAMBDA (MARRIED *X
                    *Y)                               (* FIND AT LEAST ONE
                                                      HOBBY IN COMMON,
                                                      OTHERWISE FAIL.)
           (ATTEMPT (MATCHQQ (TUPLE (CLASS *H
                                           **OTHERS)
                                    (CLASS *H
                                           **OTHEROTHERS))
                             (TUPLE (QGET (PERSON $X)
                                          HOBBIES)
                                    (QGET (PERSON $Y)
                                          HOBBIES)))
                    ELSE (FAIL CALLER))))

(PARTNERSEX
  (QLAMBDA *X                                         (* FIND THE OPPOSITE SEX
                                                      OF THE PERSON IN
                                                      QUESTION.)
            (SELECTQ (QGET $X SEX)
                     (MALE (QUOTE FEMALE))
                     (FEMALE (QUOTE MALE))
                     (ERROR "UNKNOWN SEX "))))
```

```
1  <REBOH>GENESIS.13    MON 13-AUG-73 2:25PM

(RICH
  (QLAMBDA (HAPPY =HUMAN)                                  (* TRY TO ACHIEVE A NET
                                                           WORTH GREATER THAN ONE
                                                           MILLION.)
                                                           (* IF ACHIEVABLE, THEN
                                                           THE HUMAN IS HAPPY.)
                                                           (* THIS ROUTINE NOW ONLY
                                                           MAKES A SIMPLE CHECK
                                                           AGAINST THE DATA BASE.)
           (IF (GREATERP (QGET (PERSON $HUMAN)
                               NETWORTH)
                         1000000)
               THEN ('(HAPPY $HUMAN))
             ELSE (FAIL))))

(MAKESPOUSE
  (QLAMBDA (PERSON =PERSON)                                (* TEAM MEMBER OF
                                                           $COMPUTERELATIONS.)
                                                           (* ENSURES THAT THE
                                                           SPOUSE IS NOT ALREADY
                                                           MARRIED.)
                                                           (* ASSERTS THAT THE
                                                           SPOUSE IS MARRIED.)
           (QPROG ((=SPOUSE
                    (QGET (PERSON $PERSON)
                          MARRIEDTO)))
                  (IF (NOT (EQ (QGET (PERSON $SPOUSE)
                                     MARRIEDTO)
                               (QUOTE NOSUCHPROPERTY)))
                      THEN (FAIL CALLER)
                    ELSE (QPUT (PERSON $SPOUSE)
                               MARRIEDTO $PERSON)))
)
  (LISPXPRINT (QUOTE GENESISFNS)
              T)
  (RPAQQ GENESISFNS (SETUP MAKEHAPPY HITCH CHECKAGE CHECKHOBBY
                           PARTNERSEX RICH MAKESPOUSE))
  (LISPXPRINT (QUOTE GENESISVARS)
              T)
  (RPAQQ GENESISVARS ($MARRIAGEDEMONS $COMPUTERELATIONS
                                      (P (QSETUP GENESISVARS))
                                      (P (DEFTYPE (QUOTE MARRIED)
                                                  (QUOTE CLASS]
  (RPAQQ $MARRIAGEDEMONS (CHECKAGE CHECKHOBBY))
  (RPAQQ $COMPUTERELATIONS (MAKESPOUSE))
  (QSETUP GENESISVARS)
  (DEFTYPE (QUOTE MARRIED)
           (QUOTE CLASS))
STOP
```

B. Sample GENESIS Run

Here is a session at the teletype using the functions of the GENESIS file in the QLISP system:

```
@LISP

INTERLISP-10  07-07-73 ...


HI, RENE.
←SYSIN(<SACERDOTI>QLISP.SYS)          Load in QLISP
(<SACERDOTI>QLISP.SYS;20)                  ⊛
QHELLO
←LOAD(GENESIS)                        Load file with user programs
FILE CREATED 19-JUL-73 16:44:07       Since the variable QTRACEALL is set to T,
GENESISFNS                               all QLAMBDA functions will be QTRACEd
GENESISVARS
GENESIS.;3
←$MARRIAGEDEMONS                      Net variables are treated just like
(CHECKAGE CHECKHOBBY)                      LISP variables
←PP(HITCH)                            QLAMBDA expressions are treated just like
                                           LAMBDA expressions
(HITCH
  [QLAMBDA (HAPPY ←HUMAN)  **COMMENT**     **COMMENT**     **COMMENT**
           (BIS (PERSON ←Y)
                SEX
                [@ (PARTNERSEX ('(PERSON $HUMAN]
               THEN (ASSERT (MARRIED $HUMAN $Y)
                            APPLY $MARRIAGEDEMONS)
                    (QPUT (PERSON $HUMAN)
                          MARRIEDTO $Y WRT GLOBAL APPLY
                          $COMPUTERELATIONS)
                    ('(HAPPY $HUMAN])
(HITCH)
←SETUP)                               Evaluate the user's initialization function
T
```

```
←MAKEHAPPY((ADAM SARA))                    Try to make Adam and Sara happy
HITCH:
QA4:ARG= (HAPPY ADAM)
  PARTNERSEX:                              ——Example of function trace
  QA4:ARG= (PERSON ADAM)
  (PARTNERSEX) = FEMALE                    ——Function returns 'FEMALE
  CHECKAGE:
  QA4:ARG= (MARRIED ADAM MARY)
  (CHECKAGE) = OK
  CHECKHOBBY:                              ——Function is called, but does
  QA4:ARG= (MARRIED ADAM MARY)                  not return; it caused a failure
  CHECKAGE:
  QA4:ARG= (MARRIED ADAM ALICE)
  CHECKAGE:
  QA4:ARG= (MARRIED ADAM EVE)
  (CHECKAGE) = OK
  CHECKHOBBY:
  QA4:ARG= (MARRIED ADAM EVE)
  (CHECKHOBBY) = ((CLASS SNAKE-CHARMING GARDENING VOLLEYBALL) (CLASS
GARDENING HUNTING FISHING))
  MAKESPOUSE:
  QA4:ARG= (PERSON ADAM)
  (MAKESPOUSE) = ADAM
(HITCH) = (HAPPY ADAM)
(HAPPY ADAM)
HITCH:
QA4:ARG= (HAPPY SARA)
  PARTNERSEX:
  QA4:ARG= (PERSON SARA)
  (PARTNERSEX) = MALE
  CHECKAGE:
  QA4:ARG= (MARRIED ADAM SARA)
RICH:
QA4:ARG= (HAPPY SARA)

GC: 8                                      Garbage collection of list storage
5980, 10068 FREE WORDS
(RICH) = (HAPPY SARA)
(HAPPY SARA)
FINISHED
←
```

Page Intentionally Left Blank

Annex B

THE SYNTAX OF QLISP STATEMENTS

Page Intentionally Left Blank

Annex B

THE SYNTAX OF QLISP STATEMENTS

For the syntax of INTERLISP statements see reference 3.

The representation of the syntax of the other QLISP statements is presented below. The following notation is used:

- Braces ({}) indicate that the enclosed elements may be omitted.
- Square brackets ([ ]) indicate the choice of one of the enclosed elements.
- A subscript indicates an element of a sequence.
- e indicates an expression to be evaluated.
- p-exp indicates an expression to be instantiated.
- tpl indicates an expression of STYPE TUPLE.
- var indicates a net variable.
- ctx indicates an expression that instantiates to a context.
- name indicates an expression that instantiates to a statement name.
- ind indicates an expression that instantiates to a property list indicator.
- prop indicates an expression that instantiates to a property.
- tuple, vector, bag, and class indicate expressions that evaluate to a TUPLE, VECTOR, BAG, or CLASS respectively. Note that a LISP list is treated as a tuple.

(ASSERT p-exp { APPLY [tuple | vector | bag | class] } {WRT ctx} {NAME name}
{ind_1 $prop_1$ ... ind_n $prop_n$})

(ATTEMPT e_1 {e_2 ... e_n} {THEN e_1' ... e_m'} {ELSE e_1'' ... e_k''})

(BIS p-exp { APPLY [tuple | vector | bag | class] } {WRT ctx} {NAME name}
{ind_1 $prop_1$ ... ind_n $prop_n$} {THEN e_1 ... e_n})

(CASES p-exp { APPLY [tuple | vector | bag | class] } {WRT ctx} {name name})

(CONTEXT [PUSH | POP | CURRENT] ctx)

(DELETE p-exp { APPLY [tuple | vector | bag | class] } {WRT ctx} {NAME name}
{ind_1 $prop_1$ ... ind_n $prop_n$})

(DENY p-exp { APPLY [tuple | vector | bag | class] } {WRT ctx} {NAME name}
{ind_1 $prop_1$ ... ind_n $prop_n$})

(DO tpl WRT ctx)

(FAIL { [CALLER | name] })

(GOAL p-exp { APPLY [tuple | vector | bag | class] } {WRT ctx} {NAME name}
{ind_1 $prop_1$ ... ind_n $prop_n$})

(INSTANCES p-exp { APPLY [tuple | vector | bag | class] } {WRT ctx} {NAME name}
{ind_1 $prop_1$ ... ind_n $prop_n$})

(IS p-exp { APPLY [tuple | vector | bag | class] } {WRT ctx} {NAME name}
{ind_1 $prop_1$ ... ind_n $prop_n$})

(ISNT p-exp { APPLY [tuple | vector | bag | class] } {WRT ctx} {NAME name}
{ind_1 $prop_1$ ... ind_n $prop_n$})

(MATCH e_1 e_2 {WRT ctx})

(MATCHQ p-exp e {WRT ctx})

(MATCHQQ $p\text{-}exp_1$ $p\text{-}exp_2$ {WRT ctx})

(QGET p-exp { APPLY [tuple | vector | bag | class] } {WRT ctx} {NAME name}
[ind | ind_1 $prop_1$ {ind_2 $prop_2$ ... ind_n $prop_n$}])

(QPROG args e_1 e_2 ... e_n)

(QPUT p-exp $\left\{\text{APPLY}\left[\begin{array}{l}\text{tuple}\\\text{vector}\\\text{bag}\\\text{class}\end{array}\right]\right\}$ {WRT ctx} {NAME name}

$\text{ind}_1\ \text{prop}_1\ \{\text{ind}_2\ \text{prop}_2\ \ldots\ \text{ind}_n\ \text{prop}_n\}$)

(QRETURN {p-exp})

(STYPE e)

(VAL var {WRT ctx})