August 1972 (R)

THE QA4 LANGUAGE APPLIED TO ROBOT PLANNING

by

Jan Derksen
Johns F. Rulifson
Richard J. Waldinger

Paper accepted for presentation at the Fall Joint Computer Conference, Anaheim, California, December 5-7, 1972.

Artificial Intelligence Center

Technical Note 65

SRI Project 8721

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

ABSTRACT

This paper introduces the first implemented version of the problem-solving language QA4 and illustrates the application of this language to some simple robot planning problems. Features of the language include built-in backtracking, parallel processing, pattern matching, and set manipulation. Some of these features are described in detail by close examination of the working of one of the robot problems. The QA4 language is intended for use in research aimed at the automatic verification, modification, and synthesis of computer programs and for semantically oriented theorem proving, as well as for robot planning as described in this paper.

I INTRODUCTION

This paper introduces an implemented version of a problem-solving language called QA4[2,9]* (Question Answerer 4) and illustrates the application of that language to some simple robot problems. This application is especially appropriate, because the QA4 language has features that are recognized as useful for problem-solving programs;[11] these features include built-in backtracking, parallel processing, pattern matching, and set manipulation. Expressions are put into a canonical form and stored uniquely, so that they can have property lists. A context mechanism is provided, so that the same expression can be given different properties in different contexts. The QA4 interpreter is implemented in LISP and can interface with LISP programs. The language is especially intended to be useful for research leading to program verification,[5] modification, and synthesis,[8] to semantically oriented theorem proving,[1] and to various forms of robot planning.[4]

---

*References are listed at the end of the paper.

II DESIGN PHILOSOPHY

QA4 has been designed with a specific problem-solving philosophy, which it subtly encourages its users to adopt, and which is an outgrowth of our experience with its antecedent, QA3.[6] QA3 contained an axiom-based theorem prover, which we attempted to use for general-purpose

problem solving. However, all knowledge had to be stored in the declarative form of logical axioms, with no indication as to its use. When a large number of facts were known, the knowledge could not be used effectively. The system became swamped with irrelevant inferences, even when supplied with several sophisticated syntactic strategies.

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

There are many similarities both in detail and in philosophy between QA4 and a language designed at MIT called PLANNER,[7] a subset of which has been implemented and has been very successful for expressing programs to manipulate building blocks. We have adapted some PLANNER features for our own uses, and some features shared by both languages have been arrived at independently. The fact that the same devices have been found useful by different groups of people and for diverse problem domains encourages us to believe that languages of this sort will have appeal for problem solvers in general.

III THE ROBOT PROBLEMS

We will now examine the kind of knowledge we expect a robot planner to have, and the class of problems we expect it to be able to solve. We will consider some problems of a type recently approached by the SRI robot.[4] We will then be better able to discuss the application of QA4 to this domain and the merits of the QA4 approach.

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

---

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

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

We supply the problem solver with a model or representation of the world, which includes the arrangement of the rooms and the positions of

and relationships between the objects. Furthermore, corresponding to each action the robot can take, we supply an operator, whose effect is to alter the model to reflect the changes the robot's action makes on the world. Each operator has preconditions, requirements that must be satisfied before it is applied.

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

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

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

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

IV THE SOLUTION OF THE PROBLEM OF TURNING ON A LIGHT

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

Both this subgoal and the unachieved precondition of the turnonlight operator, that the box be next to the light switch, are achieved by the pushto operator, which can move a box anywhere in the room. However, the

pushto operator still has the precondition that the robot be next to the box. This new subgoal can be achieved by the goto operator, which can move the robot anywhere around the room. The only precondition of the goto operator is that the robot be in the same room as its destination, but this condition is satisfied in our initial model, since the robot and the box are assumed to be in the same room. Thus, a solution has been found: the sequence goto the box, pushto the box next to the light switch, climbonbox, and turnonlight.

The QA4 solution to the robot problems is a direct translation of the approach of the STRIPS problem-solving system,[3] which uses the above framework. STRIPS is the problem solver that does the planning for the SRI robot. The solution of the first three problems approached by STRIPS was the first exercise for the QA4 language. The operators were encoded in the QA4 language, and the model was expressed as a sequence of QA4 statements. This package of information, with no further supervision or strategy, sufficed for the solution of the three sample problems. Finding the plans amounted to evaluating the goals expressed in the QA4 language. The solutions were found quickly and with no more search than necessary. More significantly, the operator descriptions were written quickly and are concise and fairly readable.

A. The STRIPS representation

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

STRIPS format rather closely. For more complex operators and plans, we may make use of more of the language features, as we shall see below.

B. QA4 representation

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

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

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

---

*The QA4 programs for the other operators, the precise formulation of the light switch problem, and the tracing of the solution of that problem are included in the appendix.

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

1. The Pattern

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

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

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

Patterns play many roles in QA4; they may appear on the left side of assignment statements and in data base queries. The ability to have a pattern as the bound variable part of a function gives us a concise notation for naming substructures of complex arguments. It also gives us a flexible alternative to the conventional function-calling mechanism, as we shall see.

2. Searching the data base

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

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

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

the problem. Thus, the fact that LIGHTSWITCH1 is a light switch is included in the model.

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

3. Choosing a box

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

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

4. Moving the box

The operator now insists as one of its preconditions that the chosen box be next to the light switch. For this purpose it uses the GOAL construct, a mechanism for activating appropriate functions without calling them by name. To move the box, the operator uses (GOAL DO (NEXTTO $N $M)).

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

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

To put an operator in a goal class we input (TO goal-class operator). For instance, for this example we assume that we have input (TO DO CLIMBONBOX).

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

Each of the operators has a bound variable pattern. To be applied to a goal it is not sufficient that the operator belong to the specified goal class; it is also necessary that the bound variable pattern of the operator match the expression specified by the goal statement. In this case the bound variable pattern of the pushto operator, (NEXTTO ←M ←N), matches the goal expression (NEXTTOBOX1 LIGHTSWITCH1), with M bound. Therefore, the pushto operator is activated.

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

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

The turnonlight operator requires not only that the box be next to the light switch, but also that the robot be on top of the box, before it can turn on the light. This precondition is described as

(GOAL DO (ON ROBOT $N)). Of the operators of the DO class, the only one whose bound variable pattern matches the goal expression is climbon-box, with pattern (ON ROBOT =M), so this operator is applied. The preconditions of this operator, including the requirement that the robot be next to BOX1, are already satisfied in the model. Thus, the operator can report a quick success.

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

V DESIGN PHILOSOPHY REVISITED

Although our operators are as concise as those of STRIPS, we have given them a certain amount of strategic information. For example, in turnonlight we tell the system that the box must be brought up to the light switch before the robot mounts the box, while in STRIPS the same preconditions are unordered, so the planner may investigate the ill-advised possibility of climbing the box first and moving it later.

STRIPS could have been given ordered preconditions, but its designers were more interested in the behavior of the problem solver when it had to discover the best ordering by itself. The decision about how many hints to give the operators is, we feel, primarily a matter of taste. For QA4 we prefer to give the operators as much information as possible, and risk the charge of using an ad hoc approach. We feel that this is the only way that our programs will solve interesting problems.

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

We also applied the system to the two other problems from the STRIPS paper.[4] In these problems the robot is envisioned to be in a building with several rooms and a corridor. It is asked first to push together the three boxes in one of the rooms and second to find its way from one of the rooms to another. The QA4 system solved these problems also, and it made no mistaken choices.

These problems are, of course, particularly simple. Had they been sufficiently complicated, any QA4 program would have to do some

searching in trying to find a solution. In that case, we would have had to write our operators in a somewhat different way.

VI OTHER FEATURES AND APPLICATIONS

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

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

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

theorem-proving ability would enable QA4 to construct programs in other languages, including QA4 itself.

We also plan to apply QA4 to the verification of existing programs.[3] For this application we need considerable sophistication in formula manipulation and the handling of arithmetic relations. We have introduced some new data types, sets, and bags (bags are like sets, but may have several instances of the same element), which simplify many arithmetic problems. For example, a stumbling block of earlier deductive systems has been the equality relation, which has required either a plethora of new axioms or a slightly less clumsy new rule of inference in order to describe its properties. In QA4 we simply place expressions known to be equal in the same set; the symmetric, reflexive, and transitive laws then follow from the properties of sets, and need not be stated explicitly.

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

VII PLANNER AND QA4

Let us examine the similarities and differences between QA4 and PLANNER. The two languages are quite similar in conception, and there are also more detailed parallels. Our operators in the above example

are similar to PLANNER consequent theorems. ASSERT, GOAL, and EXISTS all have their PLANNER counterparts. Both the variable prefixes and the heavy reliance on pattern matching are PLANNER features. PLANNER also has built-in backtracking.

There are some differences in detail. QA4 relies more on the use of new data types such as sets, whereas PLANNER would implement the same features by using more complex procedures. The context mechanism is unique to QA4, and a coroutine mechanism has been implemented in QA4 but is only projected in PLANNER. The PLANNER pattern matcher that has been implemented does not allow the nesting of patterns. Moreover, QA4 is intended to be modified in design and implementation, with experience. Therefore it is implemented in LISP, and we have no immediate plans to rewrite it in assembly code; PLANNER is being implemented in assembly code.

VIII IMPLEMENTATION

The BBN-LISP system[10] in which QA4 is embedded has many sophisticated debugging features, and QA4 has been designed to take advantage of such packages as the BBN editor and the programmer's assistant. At the top level, QA4 and LISP coexist side by side: Lines preceded by ! go to the QA4 evaluator rather than the LISP interpreter.

Although we have never stressed efficiency in our design, we have managed to make the implementation reasonably efficient. Our

representation of the robot problems compares favorably with that of STRIPS: The solutions to the three problems were each found in less than half a minute. The TRACE feature, whose output is shown in the appendix, is quite elaborate; we can follow the search for a solution with the appropriate degree of detail.

The ease with which it was possible to construct a robot planner on this scale gives us hope that QA4 will be an appropriate vehicle for the development of more complex problem solvers.

IX ACKNOWLEDGMENTS

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

The goals and original concepts of QA4 were formulated with C. Cordell Green and Robert A. Yates. We have benefited by discussions with and criticism from other members of the staff of the Artificial Intelligence Center at SRI, especially Bertram Raphael and the STRIPS group, Richard E. Fikes, Peter E. Hart, and Nils J. Nilsson. We have also profited from many conversations with Carl Hewitt. Nils J. Nilsson and Bertram Raphael gave us critical readings of the manuscript.

REFERENCES

1 J A C DERKSEN
The QA4 primer
Artificial Intelligence Technical Note 60
Artificial Intelligence Center
Stanford Research Institute Menlo Park California 1972

2 J F RULIFSON
QA4 programming concepts
Artificial Intelligence Center
Stanford Research Institute Menlo Park California 1971

3 T A WINOGRAD
Procedures as a representation for data in a computer program for understanding natural language
Ph.D. Thesis
Department of Mathematics
Massachusetts Institute of Technology Cambridge Massachusetts

4 R W FLOYD
Assigning meanings to programs
Mathematical Aspects of Computer Science Volume 19
American Mathematical Society pp 19-32
Providence Rhode Island 1967

5 Z MANNA AND R WALDINGER
Towards automatic program synthesis
CACM Volume 14 pp 151-165 1971

6 W W BLEDSOE R S BOYER AND W H HENNEMAN
Computer proofs of limit theorems
Artificial Intelligence Volume 3 pp 27-68 1972

7 R E FIKES AND N J NILSSON
STRIPS: a new approach to the application of theorem proving to problem solving
Artificial Intelligence Volume 2 pp 289-308 1971

8 C C GREEN
Application of theorem proving to problem solving
International Joint Conference on Artificial Intelligence
Washington 1969

9 C HEWITT

Description and theoretical analysis (using schematic) of PLANNER: a language for proving theorems and manipulating models in a robot
Ph.D. thesis
Department of Mathematics
Massachusetts Institute of Technology Cambridge Massachusetts 1972

10 B ELSPAS M W GREEN K N LEVITT AND R J WALDINGER
Research in interactive program-proving techniques
Information Science Laboratory
Stanford Research Institute Menlo Park California

11 W TEITELMAN D G BOBROW A K HARTLEY AND D L MURPHY
BBN-LISP TENEX reference manual
Cambridge Massachusetts 1972

APPENDIX

THE QA4 OPERATORS: THE GOTHRUDOOR OPERATOR

```
[RPAQQ GOTHRUDOOR (LAMBDA (INROOM (TUPLE ROBOT ←M))
          (PROG (DECLARE X K L)
                (IF (NOT $ONFLOOR)
                    THEN
                    (FAIL))
                (EXISTS (CONNECTS ←K ←L $M))
                (GOAL GO (INROOM ROBOT $L))
                (GOAL GO (NEXTTO ROBOT $K))
                (MAPC (QUOTE (TUPLE (ATROBOT ←X)
                                    (NEXTTO ROBOT ←X)))
                      $DELETE)
                (ASSERT (INROOM ROBOT $M))
                (BUILD (' (:$GOTHRUDOORACTION $K]
```

THE GOTO2 OPERATOR

```
[RPAQQ GOTO2 (LAMBDA (NEXTTO ROBOT ←M)
          (PROG (DECLARE X Y)
                (IF (NOT $ONFLOOR)
                    THEN
                    (FAIL))
                (ATTEMPT (EXISTS (INROOM $M ←X))
                         (GOAL GO (INROOM ROBOT $X))
                         THEN
                         (GO FINISH)
                         ELSE
                         (GOAL GO (INROOM ROBOT ←X))
                         (EXISTS (CONNECTS $M $X ←Y)))
            FINISH
                (MAPC (QUOTE (TUPLE (ATROBOT ←X)
                                    (NEXTTO ROBOT ←X)))
                      $DELETE)
                (ASSERT (NEXTTO ROBOT $M))
                (BUILD (' (:$GOTO2ACTION $M]
```

THE FUNCTION DELETE

```
[RPAQQ DELETE (LAMBDA ←EXP
          (PROG (DECLARE X)
                (ATTEMPT (SETQ ←X (EXISTS $EXP))
                         THEN
                         (ERASE $X]
```

THE GOTO1 OPERATOR

```
[RPAQQ GOTO1 (LAMBDA (ATROBOT ←M)
          (PROG (DECLARE X)
                (IF (NOT $ONFLOOR)
                    THEN
                    (FAIL))
                (EXISTS (DOOR INROOM $M ←X))
                (GOAL GO (INROOM ROBOT $X))
                (MAPC (QUOTE (TUPLE (ATROBOT ←X)
                                    (NEXTTO ROBOT ←X)))
                      $DELETE)
                (ASSERT (ATROBOT $M))
                (BUILD (' (:$GOTO1ACTION $M]
```

THE PUSHTO OPERATOR

```
[RPAAQ PUSHTO (LAMBDA (:EXITTO ←M ←N)
          (PROG (DECLARE X)
                (IF (NOT $ONFLOOR)
                    THEN
                    (FAIL))
                (EXISTS (PUSHABLE $M))
                (ATTEMPT (EXISTS (INROOM $M ←X))
                         (EXISTS (INROOM $N $X))
                         THEN
                         (GO FINISH)
                         ELSE
                         (EXISTS (INROOM $M ←X))
                         (EXISTS (CONNECTS $N $X ←Y)))
            FINISH
                (GOAL GO (NEXTTO ROBOT $N))
                (MAPC (QUOTE (TUPLE (ATROBOT ←X)
                                    (AT $M ←X)
                                    (NEXTTO ROBOT ←X)
                                    (NEXTTO $M ←X)
                                    (NEXTTO $X $M)))
                      $DELETE)
                (ASSERT (NEXTTO $M $N))
                (ASSERT (NEXTTO $N $M))
                (ASSERT (NEXTTO ROBOT $M))
                ($BUILD (' (:$PUSHTOACTION (TUPLE $M $N]
```

THE CLIMBONBOX OPERATOR

```
[RPAAQ CLIMBONBOX (LAMBDA (ON ROBOT ←M)
          (PROG (DECLARE X)
                (IF (NOT $ONFLOOR)
                    THEN
                    ($CLIMBOFFBOX))
                (EXISTS (TYPE $M BOX))
                (GOAL GO (NEXTTO ROBOT $M))
                ($DELETE (QUOTE (ATROBOT ←X)))
                (SETQ ←ONFLOOR FALSE)
                (ASSERT (ON ROBOT $M))
                ($BUILD (' (:$CLIMBONBOXACTION $M]
```

THE CLIMBOFFBOX OPERATOR

```
[RPAAQ CLIMBOFFBOX (LAMBDA (TUPLE)
          (PROG (DECLARE M)
                (EXISTS (ON ROBOT ←M))
                (EXISTS (TYPE $M BOX))
                ($DELETE (QUOTE (ON ROBOT $M)))
                (SETQ ←ONFLOOR TRUE)
                ($BUILD (' (:$CLIMBOFFBOXACTION $M]
```

THE TURNONLIGHT OPERATOR

```
[RPAAQ TURNONLIGHT (LAMBDA (STATUS ←M ON)
          (PROG (DECLARE N)
                (EXISTS (TYPE $M LIGHTSWITCH))
                (EXISTS (TYPE ←N BOX))
                (GOAL GO (NEXTTO $N $M))
                (GOAL GO (ON ROBOT BOX1))
                ($DELETE (QUOTE (STATUS $M OFF)))
                (ASSERT (STATUS $M ON))
                ($BUILD (' (:$TURNONLIGHTACTION $M]
```

THE FUNCTION BUILD

```
[RPAQQ BUILD (LAMBDA ←X
          (SETQ ←ANSWER (CONS $X $ANSWER]
```

THE FUNCTION SOLVE

```
[RPAQQ SOLVE (LAMBDA ←PROBLEM
          (PROG (DECLARE X)
                (SETQ ←X (REVERSE $PROBLEM))
                (RETURN (PROG (DECLARE)
                              $$X ]
```

The model of the robot world. Expressions are evaluated by the QA4 evaluator ("!") and stored in the net.

```
(DEFLIST(QUOTE(
  [SETUP (((! (TO DO CLIMBCNBOX)))
          ((! (TO DO TURNONLIGHT)))
          ((! (TO DO PUSHTO)))
          ((! (TO GO GOTHRUDOOR)))
          ((! (TO GO GOTO1)))
          ((! (TO GO GOTO2)))
          ((! (SETQ ←ONFLOOR TRUE)))
          [(! (SETQ ←ANSWER (' (TUPLE]
          [(! (ASSERT (INROOM LIGHTSWITCH1 ROOM1]
          [(! (ASSERT (INROOM ROBOT ROOM1]
          [(! (ASSERT (ATROBOT E]
          [(! (ASSERT (LOCINROOM F ROOM4]
          [(! (ASSERT (PUSHABLE BOX1]
          [(! (ASSERT (PUSHABLE BOX2]
          [(! (ASSERT (PUSHABLE BOX3]
          [(! (ASSERT (INROOM BOX1 ROOM1]
          [(! (ASSERT (INROOM BOX2 ROOM1]
          [(! (ASSERT (INROOM BOX3 ROOM1]
          [(! (ASSERT (STATUS LIGHTSWITCH1 OFF]
          [(! (ASSERT (TYPE LIGHTSWITCH1 LIGHTSWITCH]
          [(! (ASSERT (TYPE BOX1 BOX]
          [(! (ASSERT (TYPE BOX2 BOX]
          [(! (ASSERT (TYPE BOX3 BOX]
          [(! (ASSERT (AT LIGHTSWITCH1 D]
          [(! (ASSERT (AT BOX1 A]
          [(! (ASSERT (AT BOX2 B]
          [(! (ASSERT (AT BOX3 C]
          [(! (ASSERT (CONNECTS DOOR1 ROOM1 ROOM5]
          [(! (ASSERT (CONNECTS DOOR1 ROOM5 ROOM1]
          [(! (ASSERT (CONNECTS DOOR2 ROOM2 ROOM5]
          [(! (ASSERT (CONNECTS DOOR2 ROOM5 ROOM2]
          [(! (ASSERT (CONNECTS DOOR3 ROOM3 ROOM5]
          [(! (ASSERT (CONNECTS DOOR3 ROOM5 ROOM3]
          [(! (ASSERT (CONNECTS DOOR4 ROOM4 ROOM5]
          ((! (ASSERT (CONNECTS DOOR4 ROOM5 ROOM4]
```

THE 3 INITIAL PROBLEM STATEMENTS

```
  [3BOXPROBLEM (((! (SOLVE (LIST (GOAL DO (NEXTTO BOX1 BOX2))
                                 (GOAL DO (NEXTTO BOX2 BOX3]
  [GOROOMPROBLEM (((! (SOLVE (GOAL GO (ATROBOT F]
  [TURNONLIGHTPROBLEM (((! (SOLVE (GOAL DO (STATUS LIGHTSWITCH1 ON]
))(QUOTE HISTORY))
```

TRACE OF THE SOLUTION OF THE PROBLEM OF TURNING ON A LIGHT

```
(GOAL GO (STATUS (TUPLE LIGHTSWITCH1 ON)))
FAILURE
TO MATCH FAILURE PUSHTO
  LAMBDA TURNONLIGHT
  SETVALUE M LIGHTSWITCH1
  (EXISTS (TYPE (TUPLE $M LIGHTSWITCH)))
  (EXISTS (TYPE (TUPLE ←M BOX)))
  SETVALUE N BOX1
  (GOAL GO (NEXTTO (TUPLE $N $M)))
  FAILURE
    LAMBDA PUSHTO
    SETVALUE N LIGHTSWITCH1
    SETVALUE M BOX1
    (EXISTS (PUSHABLE $M))
    (EXISTS (INROOM (TUPLE $M ←X)))
    SETVALUE X ROOM1
    (EXISTS (INROOM (TUPLE GO $X)))
    (GOAL GO (NEXTTO (TUPLE ROBOT $M)))
    FAILURE
    TO MATCH FAILURE GOTHRUDOOR
    TO MATCH FAILURE GOTO1
      LAMBDA GOTO2
      SETVALUE M BOX1
      (EXISTS (INROOM (TUPLE $M ←X)))
      SETVALUE X ROOM1
      (GOAL GO (INROOM (TUPLE ROBOT $X)))
        LAMBDA DELETE
        SETVALUE EXP (ATROBOT ←X)
        (EXISTS $EXP)
        SETVALUE X E
        SETVALUE X (ATROBOT E)
        (DENY $X)
        LAMBDA DELETE
        SETVALUE EXP (NEXTTO (TUPLE ROBOT ←X))
        (EXISTS $EXP)
        FAILURE
      (ASSERT (NEXTTO (TUPLE ROBOT $M)))
        LAMBDA BUILD
        SETVALUE X (←GOTO2ACTION BOX1)
        SETVALUE ANSWER (TUPLE (←GOTO2ACTION BOX1))
      LAMBDA DELETE
      SETVALUE EXP (ATROBOT ←X)
      (EXISTS $EXP)
      FAILURE
      LAMBDA DELETE
      SETVALUE EXP (AT (TUPLE $M ←X))
      (EXISTS $EXP)
      FAILURE
      LAMBDA DELETE
      SETVALUE EXP (NEXTTO (TUPLE ROBOT ←X))
```

```
      (EXISTS $EXP)
      SETVALUE X BOX1
      SETVALUE X (NEXTTO (TUPLE ROBOT, BOX1))
      (DENY $X)
      LAMBDA DELETE
      SETVALUE EXP (NEXTTO (TUPLE $M _X))
      (EXISTS $EXP)
      FAILURE
      LAMBDA DELETE
      SETVALUE EXP (NEXTTO (TUPLE $X $M))
      (EXISTS $EXP)
      FAILURE
    (ASSERT (NEXTTO (TUPLE $M $N)))
    (ASSERT (NEXTTO (TUPLE $M $M)))
    (ASSERT (NEXTTO (TUPLE ROBOT $M)))
      LAMBDA BUILD
      SETVALUE X (PUSHTOACTION (TUPLE BOX1 LIGHTSWITCH1))
      SETVALUE ANSWER (TUPLE (PUSHTOACTION (TUPLE BOX1 LIGHTSWITCH1))
(GOTO2ACTION BOX1))
  (GOAL DO (ON (TUPLE ROBOT BOX1)))
  FAILURE
  TO MATCH FAILURE PUSHTO
  TO MATCH FAILURE TURNONLIGHT
    LAMBDA CLIMBONBOX
    SETVALUE M BOX1
    (EXISTS (TYPE (TUPLE $M BOX)))
    (GOAL GO (NEXTTO (TUPLE ROBOT $M)))
      LAMBDA DELETE
      SETVALUE EXP (ATROBOT _X)
      (EXISTS $EXP)
      FAILURE
    SETVALUE ONFLOOR FALSE
    (ASSERT (ON (TUPLE ROBOT $M)))
      LAMBDA BUILD
      SETVALUE X (CLIMBONBOXACTION BOX1)
      SETVALUE ANSWER (TUPLE (CLIMBONBOXACTION BOX1) (PUSHTOACTION
(TUPLE BOX1 LIGHTSWITCH1)) (GOTO2ACTION BOX1))
    LAMBDA DELETE
    SETVALUE EXP (STATUS (TUPLE $M OFF))
    (EXISTS $EXP)
    FAILURE
  (ASSERT (STATUS (TUPLE $M ON)))
    LAMBDA BUILD
    SETVALUE X (TURNONLIGHTACTION LIGHTSWITCH1)
    SETVALUE ANSWER (TUPLE (TURNONLIGHTACTION LIGHTSWITCH1) (
CLIMBONBOXACTION BOX1) (PUSHTOACTION (TUPLE BOX1 LIGHTSWITCH1))
(GOTO2ACTION BOX1))
  LAMBDA SOLVE
  SETVALUE PROBLEM (TUPLE (TURNONLIGHTACTION LIGHTSWITCH1) (
CLIMBONBOXACTION BOX1) (PUSHTOACTION (TUPLE BOX1 LIGHTSWITCH1))
(GOTO2ACTION BOX1))
  SETVALUE X (TUPLE (GOTO2ACTION BOX1) (PUSHTOACTION (TUPLE BOX1
LIGHTSWITCH1)) (CLIMBONBOXACTION BOX1) (TURNONLIGHTACTION LIGHTSWITCH1))
```

THE ANSWER RETURNED BY QA4

```
(PROG (DECLARE) (GOTO2ACTION BOX1) (PUSHTOACTION (TUPLE BOX1
LIGHTSWITCH1)) (CLIMBONBOXACTION BOX1) (TURNONLIGHTACTION LIGHTSWITCH1))
```