August 1971

QA4 PROGRAMMING CONCEPTS

by
Johns F. Rulifson

Artificial Intelligence Group
Technical Note 60

SRI Project 8721

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

ABSTRACT

The QA4 programming language is designed for the writing of theorem-provers, robot planners, and problem solvers. The language permits the specification of ambiguous, unorganized programs that solve problems mainly through the use of expression transformation programs that are guided by semantic and pragmatic information. This note presents an informal introduction to the unusual programming concepts used in the construction of such problem-solving programs.

I INTRODUCTION

A. Project Origin

QA4 was started by Cordell Green and Robert Yates at SRI just after Green finished his Ph.D. thesis at Stanford in 1969. His thesis was on the use of resolution based theorem-proving systems as a means to automatic question answering. Their system was named QA3[1]. It did, and still does, prove theorems in the first-order predicate calculus using resolution. This system, in fact, is the basis of the ZORBA[2] and STRIPS[3] projects currently under way at SRI.

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

B. The Language

Thus the original goal of the QA4 project was to write a theorem prover for automatic question answering. The formal language was to be far more natural than first-order predicate calculus.

This theorem prover was to perform expression transformations on concise expressions in such a way that it produced proofs with a natural style--the kind that we would accept as being intuitive and obviously dominated by the semantics of the problem. As we began to write such a theorem prover, however, we were continually confronted with the restrictions of LISP. We wanted our program to plan and reason in a common-sense way. Thus we felt that the first step was to produce some theorem-proving protocols that looked intuitive, and to be sure that these protocols could be guided by natural strategies--the kind of advice you would give students. Then we should design a system that could take such strategies and attempt to execute them. When the strategies fail, we want easy, accessible methods of adding more advice and program reorganization. We felt that the project would proceed iteratively--we would start a language and a theorem-prover simultaneously, and let each guide the development of the other.

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

C. The Problem Domain

Both the search space and the solutions for the problems we are concerned with are small. Consider an example program verification problem. Using a resolution proof method, there are over 200 individual, necessary steps in the proof of the program's correctness. By using extended omega-order logic and simplification methods the proof can be reduced to about 20 steps. Of these, 15 are obvious deductions (e.g., from A&B deduce both A and B). The remaining five steps require ingenious instantiations and use of induction. We expect a QA4 program verifier to have many special rules and detailed advice on their use. It should produce the 20-step proof with little or no wasted effort. Thus the emphasis in our language design is to permit the specification of many high level rules and strategies. We anticipate that individual strategy steps may be time-consuming, but that each step is valuable.

II EXPRESSIONS

A. Motivation

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

B. Data Structures

1. Overview

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

2. Tuples

Tuples are ordered lists; our notation is <1, 2, 3>.

3. Bags

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

4. Sets

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

C. Example

1. PLUS

Our definition of PLUS illustrates the use of these structures. PLUS is associative and commutative, so it may take a bag as its argument. QA4 always communicates with users in an infix-language. Thus a user may type 1+2+1 as a line of a program. Internally, QA4 uses a prefix representation; thus that line means +[1, 1, 2]. PLUS may not take a set as an argument, for then we would have 1 + 1 = +{1} = 1.

2. EQUAL

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

3. X+Y = Y+X

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

infix	X+Y = Y+X
initial parse	=$\{$+[X, Y], +[Y, X]$\}$
reduced standard form	=$\{$+[X, Y]$\}$

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

D. Composition

1. Canonical Forms

As expressions are composed, they are converted to a canonical form so that semantic and pragmatic properties attached to them can be associated automatically with all equivalent expressions. Composition takes place whenever a particular data structure is constructed by a program.

2. Example

For example, the process of interpreting the statement

$$X \leftarrow \{RED, BLUE, GREEN\}$$

not only assigns X to be a set, but also composes a canonical representation of the set. The composition process ensures that if the datum described by the expression (in this case, a set) has ever been previously constructed, the original value is used and no new equal but different structure is constructed.

3. Bound Variables.

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

(LAMBDA <X, Y>, (X+Y)*(Y+1))

and

(LAMBDA <U, V>, (1+V)*(V+U))

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

E. Pattern Matching

1. How It Integrates

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

2. Decomposition

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

```
{?X, ?Y, ...} ← {RED, BLUE, GREEN, YELLOW}
```

assigns one of the four color words to both X and to Y. The triple dots denote a fragment, and permit the sets on opposite sides of the arrow to be of different cardinality. Since sets are involved, X and Y may be assigned the same word or different words.

3. Retrieval

The statement

```
EXISTS {←X, RED, ...}
```

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

III CONTROL

A. Motivation

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

B. Context

1. Motivation

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

If X is a prime, then X is odd.

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

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

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

2. Example

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

```
TO PROVE X→Y where X and Y are any expressions
             with respect to context C
ESTABLISH A NEW CONTEXT C′
ASSERT X wrt C′
GOAL, PROVE Y wrt C′
ERASE C′
ASSERT X→Y wrt C
```

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

3. QA4 Contexts

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

C. Processes

1. Motivation

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

2. OR Example

Given the theorem-proving problem

PROVE A OR B

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

> Find the best part to work on, say B.
>
> Start proving it.
>
> If it progresses rapidly, keep working.
>
> If not (we may have made a mistake), save the state of this theorem-proving process, and start out on A.
>
> If A begins to look harder than B, go back to B. But now incorporate the information you have learned from working on A.

We feel concrete realizations of strategies of this type are necessary for the construction of problem solvers, and that, by using QA4, we can develop such strategies.

3. Exists Example

Another especially relevant problem is to prove

(EX(X), P(X) AND Q(X)),

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

4. Summary

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

D. Indecision

1. Valueless Variables

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

Move a block down the hall,

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

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

AT <ROBOT, ?X>

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

2. Backtracking

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

E. Iteration

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

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

or

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

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

IV ORGANIZATION

A. Review of Goals

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

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

B. Goals

1. Motivation

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

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

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

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

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

or

```
GOAL SIMPLIFY, some exp;  .
```

2. How They Work

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

```
(LAMBDA  ?X =>?Y, ...) .
```

The program also has a name, say CONDER. Now to inform the interpreter that CONDER will solve goals, we state:

```
TO PROVE USE CONDER; .
```

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

```
                    (LAMBDA X:  A&B ..,   ...)
```

and be named CONJ. When we state

```
                         TO PROVE USE CONJ;
```

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

3. No Names

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

4. Satisfied Goals

Sometimes a goal may be satisfied because an expression already exists in the data base under the current context and with appropriate properties. This corresponds to the case of

"Do I already know the solution?"

If this might be so, we can give the interpreter other programs that will test already existing expressions to see if they satisfy the goal. For example, the program named TESTIT,

```
(LAMBDA $X, (EVAL $X) = TRUE);
```

will be true only if X already has the value TRUE. We would say

```
SATISFIED PROVE USE TESTIT .
```

And now, for a PROVE goal, the interpreter first interrogates the data base to see if expressions exists that can match the goal. If some do, they will be tested by TESTIT before any solution programs are tried.

C. Pattern Predicates and Advice

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

1. Header Tests

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

```
(LAMBDA ?X => ?Y,
          (IF $X = FALSE THEN FAIL) ...)
```

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

2. Pattern Predicates

In order to make this initial test easy for the programmer, each pattern may have an associated predicate that must be true if the pattern match is to succeed. This initial test must be isolated and moved into the binding, instead of residing at the start of the program. For example,

(LAMBDA ?X ⇒ ?Y ↑ $X = FALSE, TRUE)

might be a strategy for simplifying implications. Since these predicates can examine and test any property of the variables that would be bound if the function is entered, we can now have many goal solution programs that work on the same basic structure, but eliminate themselves from possible execution.

3. Advice

As a final method of giving advice to the system, we may specify that a strategy program is to have control over the order and execution of possible goal solution programs. When these options are used, the strategy program is given the set of choices along with other necessary information. In a manner similar to co-routine execution, this strategy program works with the QA4 interpreter to order and control further executions. The strategy program may even try the solutions in parallel. This permits one to work for a while, examine the data base, shift its attention to another, and later resume the former.

D. Models and WHENs

1. Models

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

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

"Where are you?"

or

"How long have you been there?"

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

2. WHEN Statements

The GOAL mechanism is a help in implementing a problem-solver that uses operative models. But the WHEN statement is the basis of the solution. This QA4 command permits us to create a "demon." The demon is assigned a set of watching posts. For example, it is assigned to watch all expressions that match a certain pattern. When information goes past its post that matches a second pattern, and satisfies that pattern's predicate, it may take control and execute programs. In our example, we may have programs that watch operations on boxes. When things are done to the boxes, these programs modify local data, or invoke yet other demons. In this way, the main method of keeping the model up to date while taking into account complex interactions between the objects is through the use of WHEN programs.

V SUMMARY

A. Status--August 1971

Most of the statements of the QA4 language work properly. This first implementation of the language is extraordinarily general. Every possible step has been taken to be sure that, if we wish, statement operation can be easily modified. This results in slow execution time, but we feel it is worth the price. We will not know just how GOALSs, WHENs, and control structures should operate until we have successfully written some major problem-solvers. So we look at the project as a design process that should converge. We will make a first pass at the language. This will permit us to try some of the concepts and discover more precisely what the language should be. As we learn, we will change the language. And hopefully, we will uncover aspects of designing and building problem-solvers at the same time that we discover more about theorem-proving, program synthesis, and robot planning.

B. Future Work

We expect to be writing QA4 programs by October 1971. While we will undoubtedly try many small examples, the main emphasis will be on two large problem-solving programs. One is a theorem-prover that is oriented toward program synthesis and program verification. The other is a robot planner for Shakey[5]. We will spend the entire next year on these two projects, together with the continual modification of the language.

C. PLANNER

QA4 is very much like Carl Hewitt's PLANNER at M.I.T. Both projects started independently, and historically, PLANNER has had far more influence on QA4 than vice versa. It is interesting, however, that while we knew of each other's work, it was only in September of 1970 that we came to realize the remarkable similarity between our goals. On the highest level, our goals are now the same--we want to build a formal language that permits the programmer to express his intuition without destroying it in the rigor of ALGOL or LISP. On the implementation level, however, the projects diverge. A full-scale PLANNER system is being coded in machine language for the MAC-AI PDP-10. This system will eventually replace LISP as the main language for expression manipulation AI programs at MAC. Thus PLANNER is well specified, and relatively frozen when compared to QA4. Our interpreter is being coded in LISP, and our emphasis is on a flexible language that can change its form as we gain experience with it. QA4 also has a somewhat greater emphasis on the use of high-level data structures to achieve a semantic compression of the problem-solving search space.

REFERENCES

1. C. Cordell Green, "Applications of Theorem Proving to Problem Solving," International Joint Conference on Artificial Intelligence, Washington, D.C. (May 1969).

2. R. E. Kling, "A Paradigm for Reasoning by Analogy," Second International Joint Conference on Artificial Intelligence, London, England (September 1-3, 1971).

3. R. E. Fikes and N. J. Nilsson, "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving," Second International Joint Conference on Artificial Intelligence, London, England (September 1-3, 1971).

4. J. Piaget, Genetic Epistemology, Columbia University Press, New York, New York (1960), p. 14.

5. B. Raphael, "Programming a Robot," Information Processing 68, North-Holland Publishing Company, Amsterdam (1969), pp. 1575-1581.

6. C. Hewitt, "PLANNER: A Language for Manipulating Models and Proving Theorems in a Robot," Massachusetts Institute of Technology, Project MAC, Artificial Intelligence Memo 168 (August 1970).