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

Nils Nilsson

March 1976

QLISP: A LANGUAGE FOR THE INTERACTIVE DEVELOPMENT OF COMPLEX SYSTEMS

by

Earl D. Sacerdoti
Richard E. Fikes
Rene Reboh
Daniel Sagalowicz
Richard J. Waldinger
B. Michael Wilber

Artificial Intelligence Center
Stanford Research Institute

Technical Note 120

SRI Projects 8721, 3805, and 4763

The work reported herein was supported by the Advanced Research Projects Agency of the Department of Defense under Contracts DAHC04-75-C-0005 and DAAG29-76-C-0012. Additional support was provided by the National Aeronautics and Space Administration under Contract NASW-2086.

ABSTRACT

This paper presents a functional overview of the features and capabilities of QLISP, one of the newest of the current generation of very high level languages developed for use in artificial intelligence (AI) research.

QLISP is both a programming language and an interactive programming environment. It embeds an extended version of QA4, an earlier AI language, in INTERLISP, a widely available version of LISP with a variety of sophisticated programming aids.

The language features provided by QLISP include a variety of useful data types, an associative data base for the storage and retrieval of expressions, the ability to associate property lists with arbitrary expressions, a powerful pattern matcher based on a unification algorithm, pattern-directed function invocation, "teams" of pattern invoked functions, a sophisticated mechanism for breaking a data base into contexts, generators for associative data retrieval, and easy extensibility.

System features available in QLISP include a very smooth interaction with the underlying INTERLISP language, a facility for aggregating multiple pattern matches, and features for interactive control of programs.

A number of the implemented applications of QLISP are briefly discussed, and some directions for future development are presented.

I INTRODUCTION

An important byproduct of research in artificial intelligence (AI) has been the development of programming languages that permit giving instructions at a very high level to a computer. A second important byproduct has been the development of highly sophisticated, supportive interactive programming environments. Tools of this kind are very important for developing AI programs, which tend to be large, complex, and subject to frequent alteration. We believe that, as the needs of the computing community grow, and the computation speed of hardware improves, the programming tools that have been a necessity to AI will become important tools of general applicability.

This paper presents a functional overview of the capabilities and features of QLISP, one of the newest of the current generation of very high level AI languages that includes MICROPLANNER [1], SAIL [2], CONNIVER [3], POPLER [4], and others. Thus, it will serve both to introduce the language to the computing community and to briefly review the features available in the new generation of AI languages. A more extensive treatment of QLISP is available elsewhere [5].

QLISP is both a programming language and an interactive programming environment. It grew out of the QA4 language [6] that was developed at SRI from 1969 to 1972. Many of the basic concepts of the language are derived from the QA4 work. QLISP embeds an extended version of QA4 in INTERLISP [7], a widely available version of LISP with a variety of sophisticated programming aids. In

addition, it provides many new features not present in other languages.

In the following section, we will describe the language features of QLISP, with special emphasis on those not available in other languages. [Bobrow and Raphael (8) give a comparative description of a number of these languages.] Then we shall describe the programming environment provided by QLISP and the underlying INTERLISP. Finally, we shall give some examples of the ways in which the language has been used to create complex software systems.

II LANGUAGE FEATURES

This section will discuss the more notable features of the QLISP language. Most of these are derived from features present in QA4. Some are derived from other languages. Most have been extended for greater ease of use, compatibility with the underlying INTERLISP language, or increased generality.

A. Data Types

QLISP provides a very rich set of data types and facilities for manipulating them. In addition to the range of types provided by INTERLISP (including numbers, arrays, strings, and list and binary tree structures), QLISP provides data of type tuple, vector, bag, and class.

A tuple is similar to a LISP list, but can be accessed via

associative retrieval as described in Section II-B below. A vector is like a tuple, but is treated somewhat differently when evaluated.

A bag is a multiset, an unordered collection of elements that may be duplicated. For example, (BAG A A B C) is equivalent to (BAG A C B A) but is different from (BAG A B C). Bags are particularly useful for describing the argument lists of associative commutative relations. For example, if we defined the relation PLUS to take a bag as its argument, then the expressions (PLUS A A B C) and (PLUS A C B A) (which would both be stored internally as (PLUS (BAG A A B C))) would be equivalent by definition.

A class is an unordered collection of elements, without duplication. For example, (CLASS A A B C) is equivalent to (CLASS C B A).

B. Associative Data Base

Expressions composed of any of the data types mentioned above may be placed in a data base. The data base is designed for **associative retrieval**, the fetching of data by content rather than by name or address. A request for an item of data may specify values for any of its constituent elements, leaving the rest to be matched by the values in the retrieved item. The data base is maintained in the form of a discrimination net, a tree-like structure in which the nodes represent tests to apply to an expression, and the branches represent the values returned by the tests. In general, these tests are set up to find the first difference, scanning left to right, between two expressions.

C. Canonical Representation of Expressions

By storing all data in a common discrimination net, QLISP can represent equivalent expressions uniquely. In the QLISP net, only one instance of an expression may occur. Before an expression is entered into the net, it is transformed into a canonical form. A new datum will not be created if the expression already occurs in the net. Thus, continuing our example about the PLUS relation, (PLUS A A B C) and (PLUS A C B A) are not only equivalent; they are exactly the same pointer into the data base.

D. Property Lists

Arbitrary expressions are represented uniquely in QLISP, just as atoms are represented uniquely in LISP. Therefore it is possible to assign properties to QLISP expressions in the same way as LISP atoms. For instance, we may execute the command

(QPUT (PLUS A B (MINUS A)) SIMPLIFIESTO B),

which will put the value B under the indicator SIMPLIFIESTO in the property list of the expression (PLUS A B (MINUS A)). If this expression, or any equivalent expression (such as (PLUS B (MINUS A) A)), is ever encountered again, we can look on its property list and find a simplification for it.

One particular indicator on the property lists of expressions is used to represent truth value. When this indicator, MODELVALUE, has a value T, the system interprets that expression to be "true." Similarly, a value of NIL represents a "false"

expression. Special statements exist for manipulating this particular property. For example, the statement

(ASSERT (AT SRI MENLO-PARK))

would simply place the attribute-value pair (MODELVALUE T) on the property list of the tuple (AT SRI MENLO-PARK).* The semantics of the statement is that SRI is in Menlo Park. Similarly, the statement

(IS (AT ←THING MENLO-PARK))

would cause a search of the data base for something that was known (i.e. was in the data base with MODELVALUE equal to T) to be in Menlo Park.

E. The Unification Pattern Matcher

An important activity in AI programs is the construction, modification, and analysis of complex symbolic expressions. The most powerful tool for this is a pattern matcher, an algorithm that allows one expression to be used as a template to break up another expression into components. QLISP extends this facility by providing a unification pattern matcher in which each of two expressions may act as templates for the other.

Some examples at this point are appropriate. The QLISP statement MATCHQQ invokes the pattern matcher directly. The statement

(MATCHQQ (←X ←Y) (A B))

---

* This paper will avoid almost all need for the reader to cope with QLISP-specific syntax. It suffices to say that in QLISP statements, the elements of expressions are presumed to be constants unless identified as a variable by the prefix ← or $. The ← prefix indicates that the variable is to be assigned a new value; the $ prefix indicates the previous value of the variable.

will match X to A and Y to B. The statement

(MATCHQQ (←X ←X) (A B))

will fail, since X cannot be bound simultaneously to A and B. The statement

(MATCHQQ (A ←X) (←Y B))

will match X to B and Y to A. The statement

(MATCHQQ (A (B ←X) ←Y) (←X ←Z (A (B C))))

will match X to A, Y to (A (B C)), and Z to (B A).

The QLISP pattern matcher is based on an extended unification algorithm that can deal with the variety of data types available in the language. The matcher is not complete for complex expressions containing bags and classes. However, it is adequate for the kinds of expressions that are almost always used. Pattern matching is used in QLISP for several central purposes. It is used to bind variables and decompose expressions, as we have mentioned. It is used to control associative retrieval. It is also used to invoke functions for specified purposes, as we will now show.

F. Pattern-Directed Function Invocation

Many of the AI languages provide a feature, first proposed by Hewitt [9], whereby functions can be invoked not only by naming them, but also by checking to see if they are appropriate for a given argument. This check is performed by matching a pattern associated with each function with the given argument. For example, we might write some functions such as the following for an algebraic

simplifier:*

```
PLUSSINGLE:     (QLAMBDA (PLUS ←X) $X)

PLUSZERO:       (QLAMBDA (PLUS 0 ←←X) ('(PLUS $$X)))

PLUSMINUS:      (QLAMBDA (PLUS ←X (MINUS ←X) ←←Y) ('(PLUS $$Y)))
```

The PLUSSINGLE function says: given an argument of the form PLUS followed by any single element, return that single element. The PLUSZERO function says: given an argument of the form PLUS followed by any number of elements, one of which is 0, return the form PLUS followed by all the other elements of the argument.

At the user's option, if a function's pattern can match an argument in more than one way, all possible matches may be attempted in turn. When one match leads to a failure, an alternative match is attempted. The function itself will not fail until all possible matches have been tried. For example, the following program will find two friends of JOE who are father and son:

```
(QLAMBDA (FRIENDS JOE (CLASS ←F ←S ←←REST))
         (IS (FATHER $S $F))
         BACKTRACK)
```

The program will cycle through all pairs of elements from the class of JOE's friends and see if one is the father of the other.

---

* The doubled prefixes (e.g. $$) indicate that the variable refers to a fragment of the expression containing it rather than a single element. The quote mark (') indicates that the following expression is to be instantiated (following the semantics of QLISP) rather than evaluated (following the semantics of LISP).

G. Teams of Functions

Functions to be invoked by pattern are typically intended for application toward a specified purpose. Some functions are to be used for consequent reasoning: when a particular consequence or goal (characterized by the function's pattern) is desired, invoke this function to achieve it. Some functions are to be used for antecedent reasoning: when a particular antecedent condition (e.g. an assertion in the data base) characterized by the function's pattern occurs, invoke this function to cause further effects on the data base.

Typically in languages that use pattern-directed function invocation, many of the consequent functions are tried when a goal is to be achieved, and all the antecedent functions are tried when the data base is updated. Only the ones that have a pattern that matches the goal or assertion will actually be invoked, but a great deal of overhead must be expended to attempt to match the patterns of all functions.

This practice is inefficient since many functions may already be known to be inappropriate, and yet their patterns will all be checked. QLISP provides a feature whereby, with each of many kinds of statements that can invoke functions by pattern, a so-called team of functions can be specified from which applicable ones may be drawn. So, in our simplification example, we could cause one simplification to occur with a statement that calls for consequent reasoning:

```
(CASES (PLUS A 4 (MINUS A)))
       APPLY (PLUSSINGLE PLUSZERO PLUSMINUS ... )) .
```

The list after the keyword APPLY is the team of functions associated with the particular CASES statement. The system will attempt to match the patterns of only these functions with the particular PLUS expression.

Similarly, a team of functions may be specified with any ASSERT, DENY, DELETE, or QPUT statement to perform antecedent-type activities. For example, in a computer system modelling the operation of a library, a team of functions might be associated with assertions that modelled a book being checked out. These functions might assert that the book was due three weeks from the current date, update a count of books in circulation, or even cause the original assertion to fail and appropriate other action to be modelled if the person checking out the book had overdue books outstanding. This activity could be initiated by a QLISP statement of the form:

```
(ASSERT (CHECKEDOUT (The Odyssey) James.Joyce (4 JAN 1918))
        APPLY $LIBRARYFNS)
```

where $LIBRARYFNS was bound to the list of relevant antecedent functions.

H. Contexts

The previous discussion has presumed that all expressions were stored in a single, monolithic data base. In fact, the data base is factored into different segments, called **contexts**. Contexts may be thought of as corresponding to the block structure of ALGOL-like languages. Whenever a QLAMBDA function or a user-defined block

is entered, the current context is set to be descendent of the previous context. Variable bindings and assignment of properties (including, in particular, truth values) to expressions that are local to a context are perceivable only from that context or some descendent. Thus, contexts may be regarded as particular viewpoints of the data base.

In addition to a default structuring of contexts based on the structure of the flow of program control, QLISP provides facilities for manipulating contexts explicitly. For example, to prove a proposition of the form:

(P or Q) implies R,

one could set up two parallel contexts with P true in one and Q true in the other, and try to prove R in both contexts, as suggested in Figure 1.

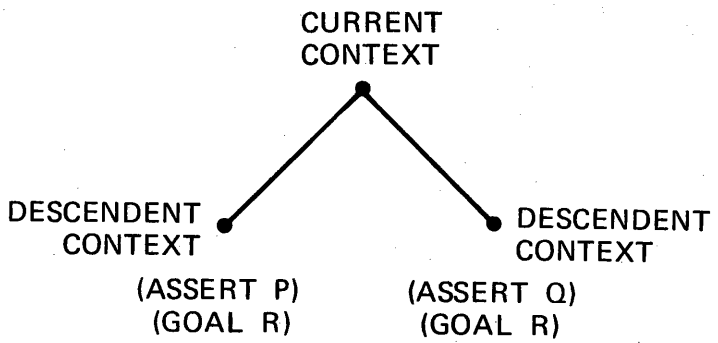

FIGURE 1 USING CONTEXTS TO PROVE A DISJUNCTION

Contexts are actually constructed from more elementary entities, which we shall call **contags**, for want of a better term.

Contags, which are similar to the "situation tags" of PLASMA [10], correspond to particular points in time in the evaluation of a program (typically when QLAMBDAs or blocks are entered). A context is an ordered list of contags, typically corresponding to the stack of active function and block invocations. For users with sophisticated needs for data base manipulation, we have provided a set of QLISP statements that permit them to construct their own contexts, or viewpoints of the data base, from the underlying contags. These statements allow the creation of a context that is a descendent of a number of independent contexts, a context that is the subset of a given context not retrievable from a second context, and a context that revises a given context to appear as if it were a descendant of another arbitrary context.

I. Generators

The data retrieval statements of QLISP are designed to find a single instance of a given pattern. To cause the pattern matcher to continue its search and obtain other such instances, a user's program must return to the query statement via the backtracking mechanism (i.e. by failing).

To allow a more natural and inexpensive method of retrieving multiple instances of a pattern, we have extended the CONNIVER [3] approach of using **generators**. For example, the IS statement that was introduced in Section II-C specifies the retrieval of one instance of a given pattern. There is a generator version of

the IS statement called GEN:IS that finds multiple true instances of a given pattern. Each time a statement such as GEN:IS is called, it produces a number of instances matching the pattern. These expressions are put on a "possibilities list" along with a "tag" that indicates how the generator can be restarted when more instances are requested, and this possibilities list is returned by the generator as its value.

If the function TRY:NEXT is called with a possibilities list as an argument, it will remove the first instance from the list and return it as its value. If the list contains no more instances, the tag is used to restart the generator. Since calls to TRY:NEXT can be made from anywhere in a program, this form of generator separates the retrieval of data elements from the processing that is done on them in a way that is not possible in a strict backtracking regimen.

The generator retrieval statements are implemented using INTERLISP FUNARGs. A FUNARG is a data object that conceptually represents a copy of a function together with that copy's private data environment.

J. Extensibility

QLISP statements that are not part of the underlying INTERLISP language are processed by the INTERLISP error handling mechanism, as will be explained below. User-oriented tools for accessing the LISP translation mechanism are provided so that new

QLISP-like statements can be defined easily. Once the statements have been defined, they are treated by the interpreter and compiler exactly like other QLISP statements. Typically, the extension facility has been used to provide alternative control structures for invoking the standard QLISP statements, or to provide special syntax for user-defined QLAMBDA functions.

III SYSTEM FEATURES

QLISP is more than just a programming language; it is an interactive programming environment for the development of very complex collections of software. In this section we shall discuss the major features of this environment that are unique to QLISP.

A. Integration with INTERLISP

The major advantage of QLISP as a programming environment, as compared with other new AI languages, is its ease of use. It is easier to edit functions, create symbolic files, trace execution paths, break into computation paths, and debug programs in QLISP. This is primarily due to the choice of INTERLISP as the host language for QLISP, and the care that has been taken in the implementation of QLISP to preserve the many supportive features of INTERLISP.

QLISP is implemented through the error handling mechanism of INTERLISP. A valid LISP expression will never be seen by the QLISP processor. Thus, programs or portions of programs that use only LISP constructs will run as fast in QLISP as in INTERLISP.

When the INTERLISP interpreter encounters an ill-formed LISP expression, it calls an error routine that in turn invokes an error analyzer. If the expression is recognized as a valid QLISP form, it is translated to an equivalent LISP form that is returned to the interpreter for evaluation. The translation is stored with the original expression so that the translation need be done only once.

A similar mechanism causes QLISP code to be translated into equivalent LISP code when it occurs within a function being compiled. Since the translation occurs at compilation time, the QLISP interpreter need never be invoked at all when running compiled QLISP code.

B. Aggregation of Pattern Matches

The "apply team" mechanism provides a good means of reducing the number of unneeded pattern matches during pattern-directed function invocation. However, there may still be a lot of wasted effort as the function invocation mechanism attempts to match each pattern in turn to the argument. For example, the simplification functions described in Section II-F all begin with PLUS. They might even be segregated into a specific team of functions to simplify expressions begining with PLUS. And yet every pattern will be matched against the argument, and the matcher will succeed as least as far as matching up the PLUS's.

An option is available to allow the patterns of QLAMBDAs to be aggregated together in a tree structure. For example, the tree

for the simplification functions listed in Section II-F appears in Figure 2. A single operation against the tree can determine the set of all the QLAMBDA functions that are good candidates to successfully match a given argument. (The tests that are applied are cruder than those applied by the pattern matcher itself, so that the set of functions may contain some whose patterns will not actually match when the matcher is invoked.) This set can then be intersected with the particular "apply team" to determine which functions to invoke.

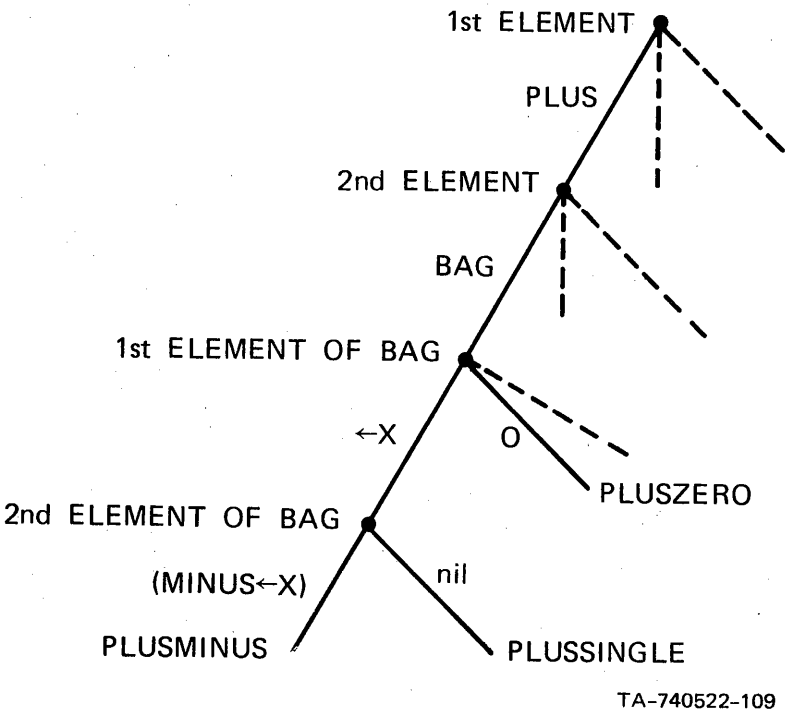

FIGURE 2 PATTERN SELECTION TREE FOR SIMPLIFICATION FUNCTIONS

The tree structure used for this aggregation is actually the discrimination net of the associative data base. For "apply

teams" of more than about fifteen functions, this feature provides significant efficiencies.

C. Interactive Program Control

Since the QLISP backtracking mechanism is implemented using INTERLISP's error facility, there are a number of ways in which the standard INTERLISP interactive facilities will not work properly. For example, the INTERLISP function tracing facility is implemented as "break the computation, then print, then continue." But INTERLISP errors, which are generated by QLISP to cause backtracking, are trapped at a break. The solution we adopted to this particular quandary was to implement a QTRACE facility that did not generate a break when it printed information about a function invocation. Similar care was taken with breaks in computation, the package for manipulating symbolic files, and many other system components to allow a QLISP user to believe that the total system was behaving exactly as the underlying INTERLISP would.

IV APPLICATIONS

To provide some perspective on the utility of QLISP, we will briefly describe some of the applications implemented with it. The common characteristics of these applications are that the programs and the concepts underlying them could not be specified without a sustained cycle of:

1) programming the best current ideas about what the program should be doing,

2) observing the program's behavior, and then

3) modifying or extending the ideas.

A. Program Verification

The first major QLISP program was the program verifier of Waldinger and Levitt [11]. The verifier was originally written in QA4. The program includes over 100 functions each encapsulating a specialized piece of knowledge about the semantics of the language of the programs being verified. The QLISP version ran about 30 times faster than the original QA4 program.

B. Automatic Programming

Subsequent work by Waldinger has used QLISP for the generation of simple programs from output specifications. This work makes strong use of the unification feature of the pattern matcher to combine the knowledge that is distributed in various QLAMBDA functions. For example, one function may say, in effect, to produce a list with some X as its CAR, perform (CONS X something), where the something is unspecified. Similarly, another function may say, to produce a list with some X as its CDR, perform (CONS something X), where the something is again unspecified. Therefore, if the system were given the goal of producing a list with A as its CAR and B as its CDR, the first function would return (CONS A something), the second would return (CONS something B), and by unifying these results, the system could produce the correct code (CONS A B).