



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

Wils Wilsson

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 DAHCO4-75-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

```
(OPUT (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) SX)

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 $$ $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).

